Machine Learning Engineer Nanodegree

Capstone Project

Anthony Chi
November 9th, 2017

I. Definition

Project Overview

I want to be a real estate investor, so I thought working on a real estate related project would be beneficial. Since I haven't been in the ML field for too long, I picked a project from kaggle: House Prices: Advanced Regression Techniques. Link for this competition: https://www.kaggle.com/c/house-prices-advanced-regression-techniques Armed with my ML knowledge and 80 features in the provided dataset, I have achieved reasonable errors in my housing price predictions. Through this exercise, I gained intuition on factors that impact house prices, worked through difficulties on datasets, and experienced with different regression models.

Please note that the Ames Housing dataset was complied by Dean De Cock for use in data science education. This is a modernized and expanded version of the often cited Boston Housing dataset. Please use the link to access the Journal of Statistics Education article published in 2011, detailing the Ames Housing dataset and Regression Project:
https://ww2.amstat.org/publications/jse/v19n3/decock.pdf

Problem Statement

The goal of this exercise was to predict house prices using available features. This was a supervised regression exercise that used features (numerical and nominal) to predict price (numerical continuous). Kaggle suggested Random Forest and Gradient Boosting as ways to tackle this exercise. However, the exercise was much more comprehensive than plugging train and test datasets into sklearn. There were data explorations and transformations, including PCA, grid search cross validation, and even MLP and deep learning components.

To complete the exercise, here's the steps that I took:

- Explore data – I explored how the data was distributed by finding the max, min, median, mean, skew, and histogram. I also looked at how the features correlated by using the pandas.corr() and seaborn.heatmap() functions.
- Clean data – I explored how each feature was presented, and how it related to the target. I also determined how to manage outliers and missings data. Visualization was also achieved to make more compelling stories.

    - Outliers: Mainly used transformations. They should only be removed unless they were recorded erroneously, or created a significant association.
    - Missing data: I removed missing data only when there's no meaning to the features, or when the majority of the feature content were the same. I leveraged the data dictionary to impute the supposedly missing values.
- Prepare data – I used sklearn.model_selection.train_test_split to randomly split the data into train and test sets.
- Feature treatments – with many categorical variables, one-hot was leveraged. With 292 features, PCA and feature engineering/transformation might be necessary.
- Model selection – given this is a regression exercise, I experimented with different modeling algorithms with different data inputs (data cleaning and feature treatments). To optimize each model's hyperparameters, I used grid search in choosing hyper parameters. I compared each optimized model and selected a final model. The model's output on the provided test dataset will be submitted to kaggle for evaluation. Here were some models that I considered:
    - DecisionTreeRegressor – use GridSearchCV, experimented with different depths and cross-validation to optimize the model performance.
    - RandomForestRegressor – use GridSearchCV, experimented with different depths and cross-validation to optimize the model performance
    - AdaBoostRegressor – use GridSearchCV, experimented with different loss and cross-validation to optimize the model performance.
    - KNeighborsRegressor – use GridSearchCV, experimented with different neighbors, algos and cross-validation to optimize the model performance.
    - GradientBoostingRegressor – use GridSearchCV, experimented with different max_depth, lossand n_estimators and cross-validation to optimize the model performance.
    - MLPRegressor – use GridSearchCV, experimented with different hidden_layer_sizes, activation, learning_rate, momentum.
    - KerasRegressor – use kfold, experimented with compile optimizers, activation, nb_epochs and batch_size, number of layers, etc.
    - Other possibilities: XGBoost, Lasso, or ENet

Metrics

The evaluation metric was the root mean square error (RMSE), which was represented by:

$$\text{RMSD} = \sqrt{\frac{\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}{n}}$$

RMSE was a standard evaluation method for modeling exercises with continuous target. Similar to the mean absolute error (MAE, another popular metric in regression), both metrics ranges from 0 to infinite, indifferent to direction of errors, and negatively-oriented. However, RSME gave relatively high emphasis to errors with outliers. I also thought about R2 (coefficient of determination) as our metric. R2 was a metric that measures how the model explain the variability in the data; however, R2 only ranged from 0 and 1 and could be difficult to interpret in a price prediction exercise. Eventually, I took kaggle's recommendation and went with RMSE as my metric.

I split the provided dataset into train, validation and test sets. I chose the modeling hyperparameters based on how the root mean squared error performed on train and validation datasets. After optimizing the hyperparameters, I chose the models based on test data performance. The optimization was accomplished by GridSearchCV and k-fold cross validation.

I also timed the modeling process as duration was an important evaluation metric for model performance.

II. Analysis

Data Exploration

The dataset for this exercise originated from the Ames City Assessor's Office. Variables that required special domain knowledge and variables that were present for weighing and adjustment purposes were removed. The data contained 79 features (23 nominal, 23 ordinal, 14 discrete and 20 continuous), 1 target, and 1459 observations. Since sensitive information, such as address or zip codes, were never present, it would be difficult to improve model performance by introducing external data and joining with the existing dataset. Please refer to the attached features excel file (Features.xlsx → Features Dictionary).

The goal of the exercise was to predict SalePrice, which was the target in our given dataset. It was a continuous variable and described the final sales price of the property. I explored all 79 features (23 nominal, 23 ordinal, 14 discrete and 20 continuous) and their potentials to predict SalePrice. From my initial exploration, features such as LotArea was highly correlated with the SalesPrice. Here were some thoughts on how variables in different categories were treated:

Variable Type	**Nominal**	**Ordinal**	**Discrete**	**Continous**
Example Feature	Heating (type of heating)	OverallQual (rating for the material and finish of the property)	FullBath (number of full bathrooms in the property)	LotArea (property plot size)
Potential Treatment	Onehot encoding	Tested whether transformation, standardization and PCA improved model performance	Tested whether transformation, standardization and PCA improved model performance	Tested whether transformation, standardization and PCA improved model perfor

For each numeric features, I recorded the following statistics: % missing, % zeros, min, max, median, mean, skew, and kurtosis. Please find the complete list in the attached excel file (Features.xlsx →

Features Summary). I also transformed features to see if skew and outliers improved. It turned out that several features' distribution improved. Here are some examples of the features and my comments:

Features	% Missing	% zeros	Min	Max	Median	Mean	Kurtosis	Skew	Trans Kurtosis	Trans Skew	Comment
Id	0.00%	0.00%	1	1,460	731	731	-1.207	0.016	5.111	-1.938	Irrelevant, delete
BsmtFinSF1	0.00%	31.60%	0	5,644	384	444	12.867	1.817	-1.385	-0.761	Distribution improved after transformation
OverallQual	0.00%	0.00%	1	10	6	6	0.05	0.191	3.219	-0.816	Distribution didn't improve after transformation

As for the categorical features, % missing and % zeros were recorded. These features were treated with one-hot.

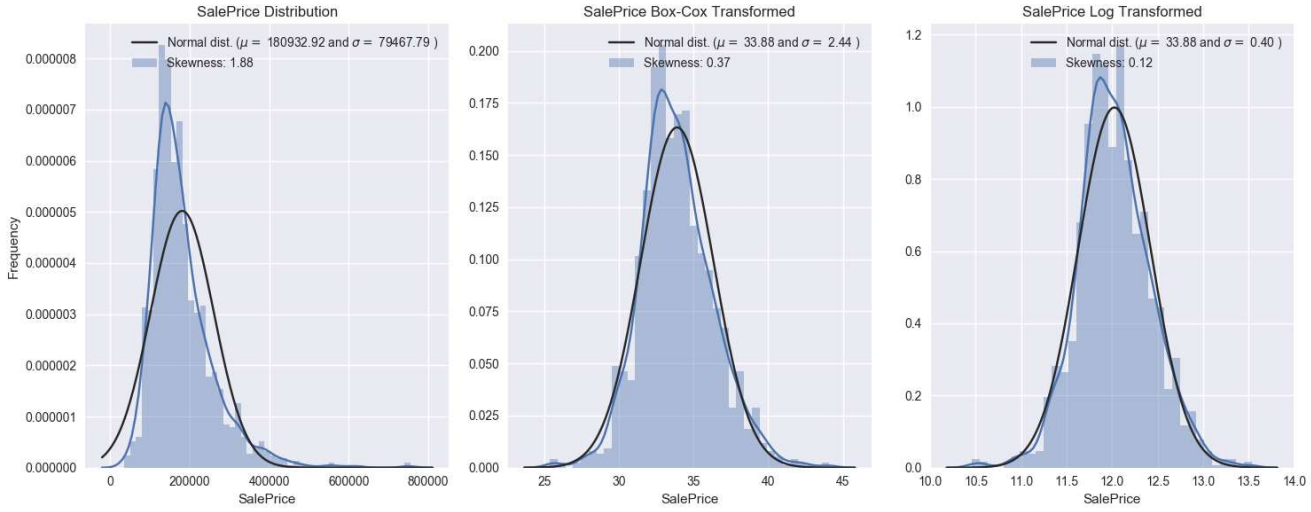
Please note that there are many features with large percentage of missing values, namely LotFrontage (17.9%), Alley (93.9%), FireplaceQu (46.9%), PoolQC (99.6%), Fence (81.7%), and MiscFeature (96.2%). I interpreted the data dictionary to my best ability and imputed the missing values.

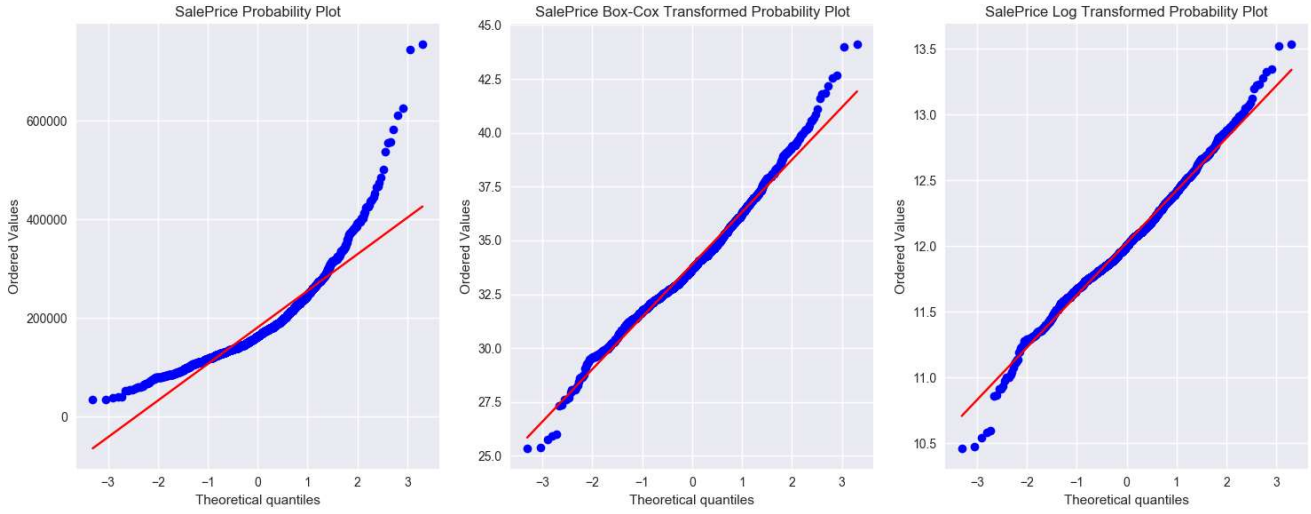
Because data for this exercise came from kaggle, I used the given training dataset as my train, validation, and test datasets. I split the data into train vs. test (80:20, random split). During the GridSearchCV process, I then specified 20% of data to be used for validation. Note that train and validation metrics were used for model optimization, while test metric was used to select a best performing model.

Exploratory Visualization

Target

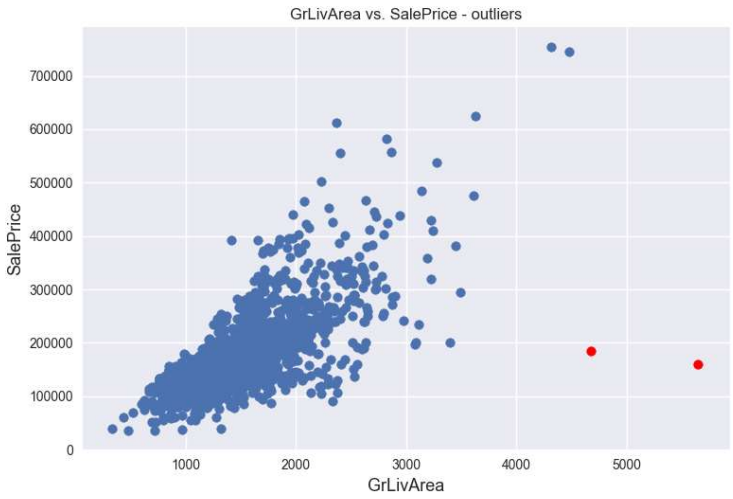
For the target (SalePrice), I performed transformations to see if distribution would improve – turned out log transformation returned the optimal distribution.



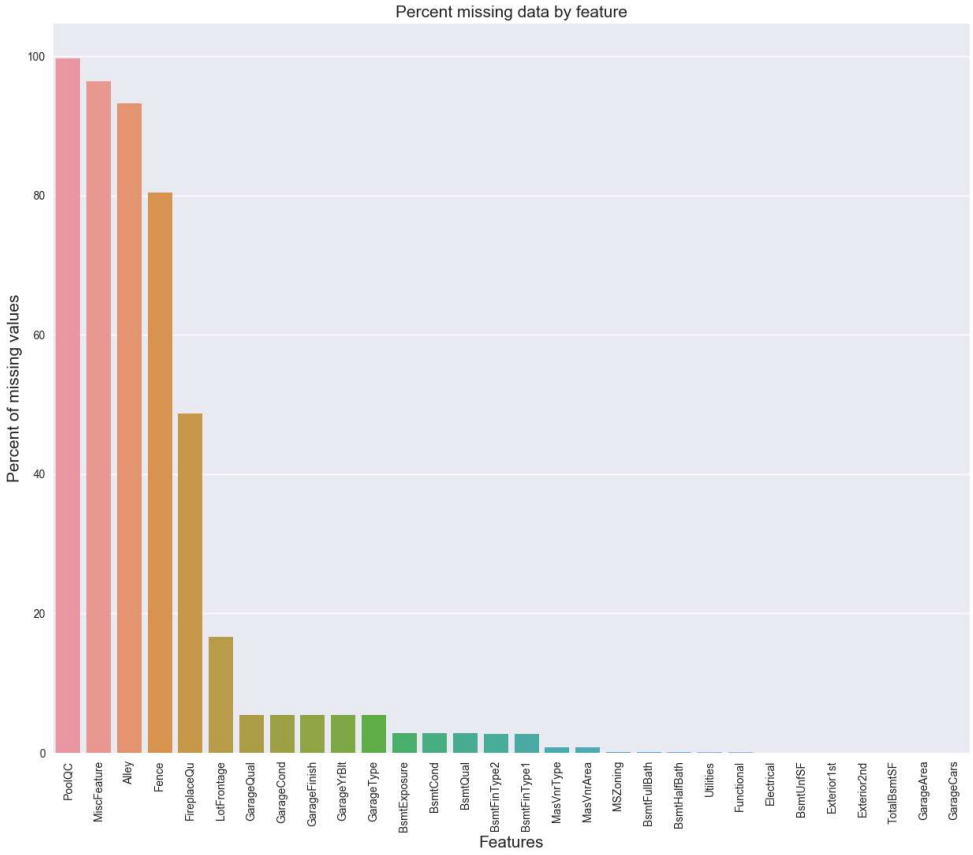


Outliers

Scrubbing through feature distributions, I found that GrLivArea was the only feature with data points that deviated from the general trend of feature vs. target. I removed these two data points, where GrLivArea > 4,000 and SalePrice < 200,000.

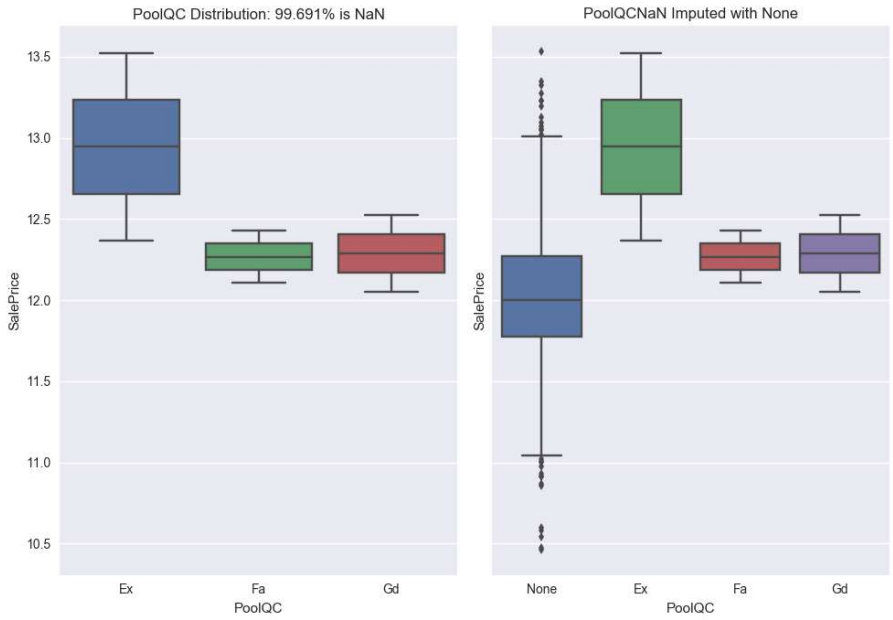


Missing Data

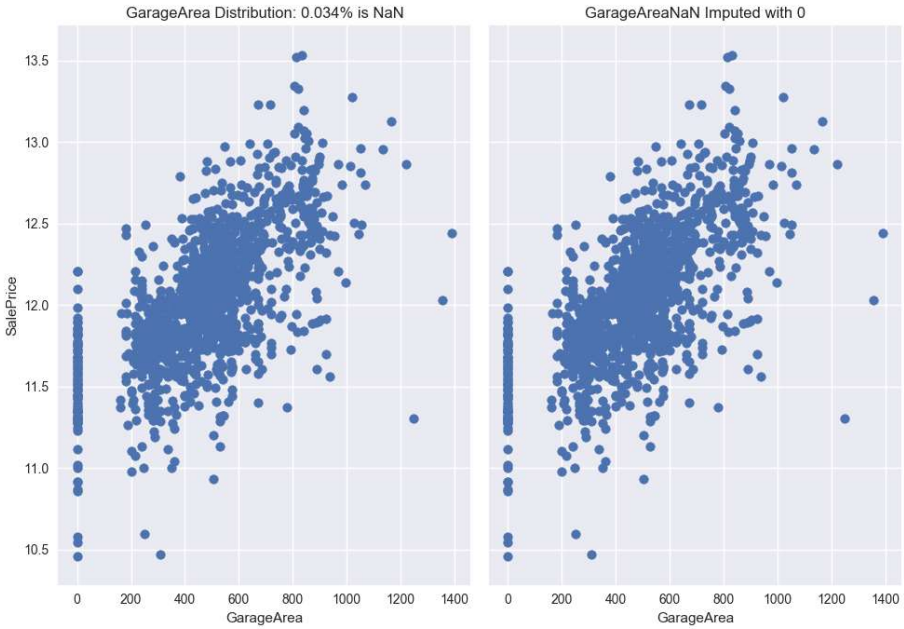


There were 36 features that contained missing data. Out of those, 4 features had missings exceeded 80%. There were several treatments for the missing values:

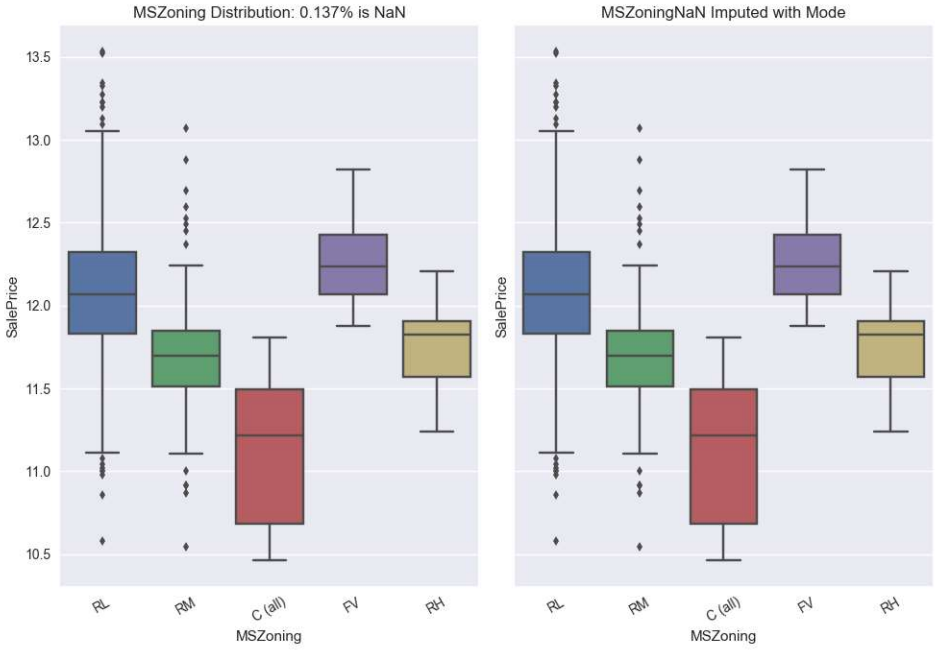
1. Missings replaced by None – these were categorical features. According to the data dictionary, missing values should be interpreted as 'None'. Example: PoolQC, Fence



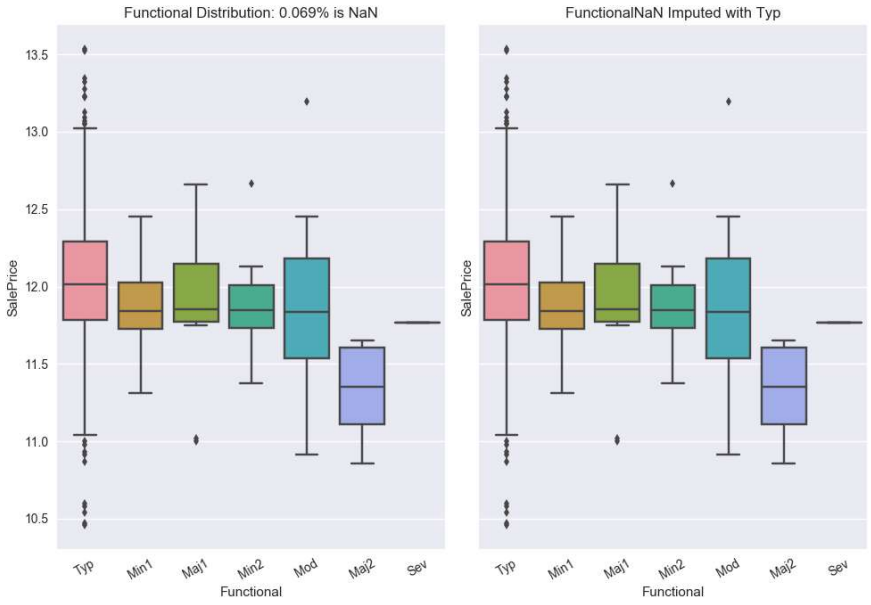
2. Missing replaced by zero – these were numerical features. According to the data dictionary, missing values should be interpreted as zeros. Example: GarageArea, BsmtFinSF1



3. Missing replaced by mode – these were features that should receive the most common values by intuition. Example: MSZoning, Electrical



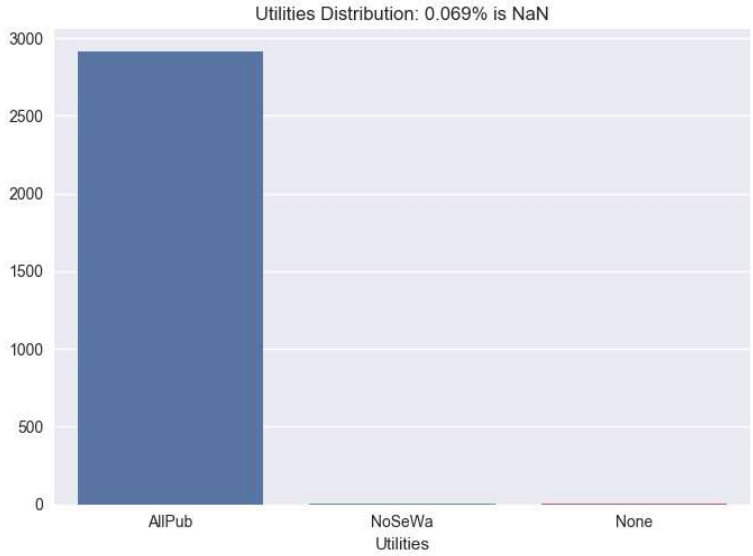
4. Missing replaced by a specific value – the missings for the Functional feature should be ‘Typ’, indicating typical, by intuition.



5. Missing replaced by median – these were features with values that were dependent on other features. Example: LotFrontage (homes in the same neighborhood should have similar LotFrontage, which implied missing LotFrontage should take on median values of homes in the same neighborhood)

Delete Feature

Meaningless features, such as Id, which indicated the record number in the dataset, should be removed from the dataset before feeding the dataset into the models. Features such as Utilities was also removed due to lack of variance in its values.



Feature Types Transformed

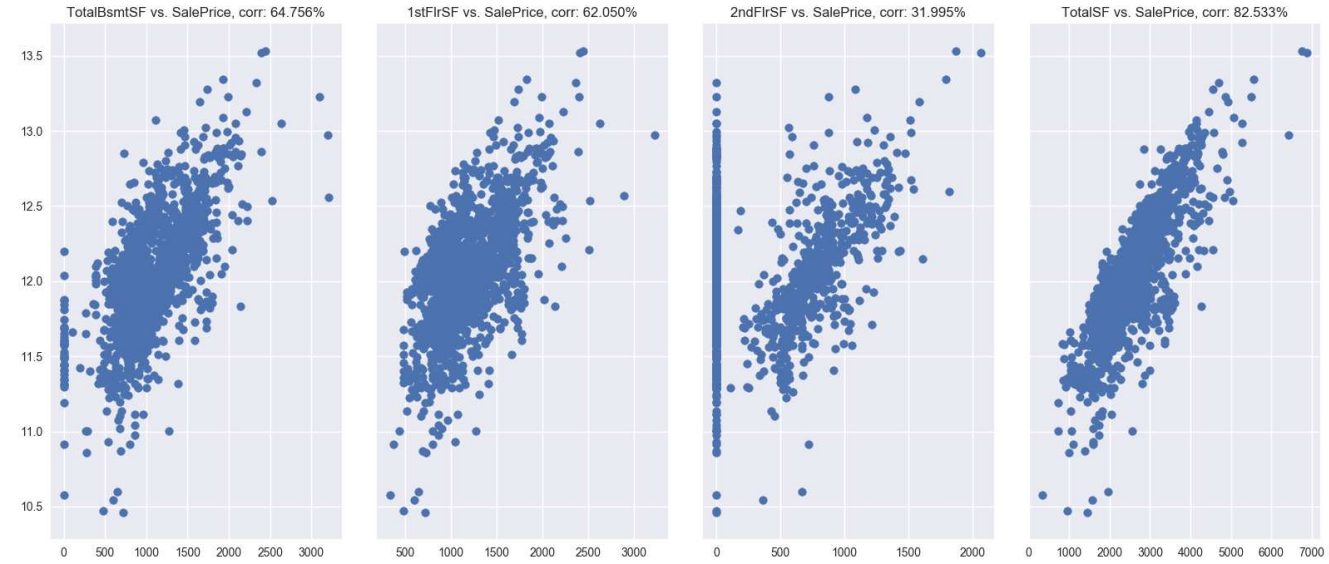
Some numerical features were transformed into categorical features because they may be more appropriate as categorical features. Example: OverallCond, MoSold

One-hot Features

Categorical features that might be correlated with the targets were transformed to one-hot features using sklearn.preprocessing.LabelEncoder().

New Features created

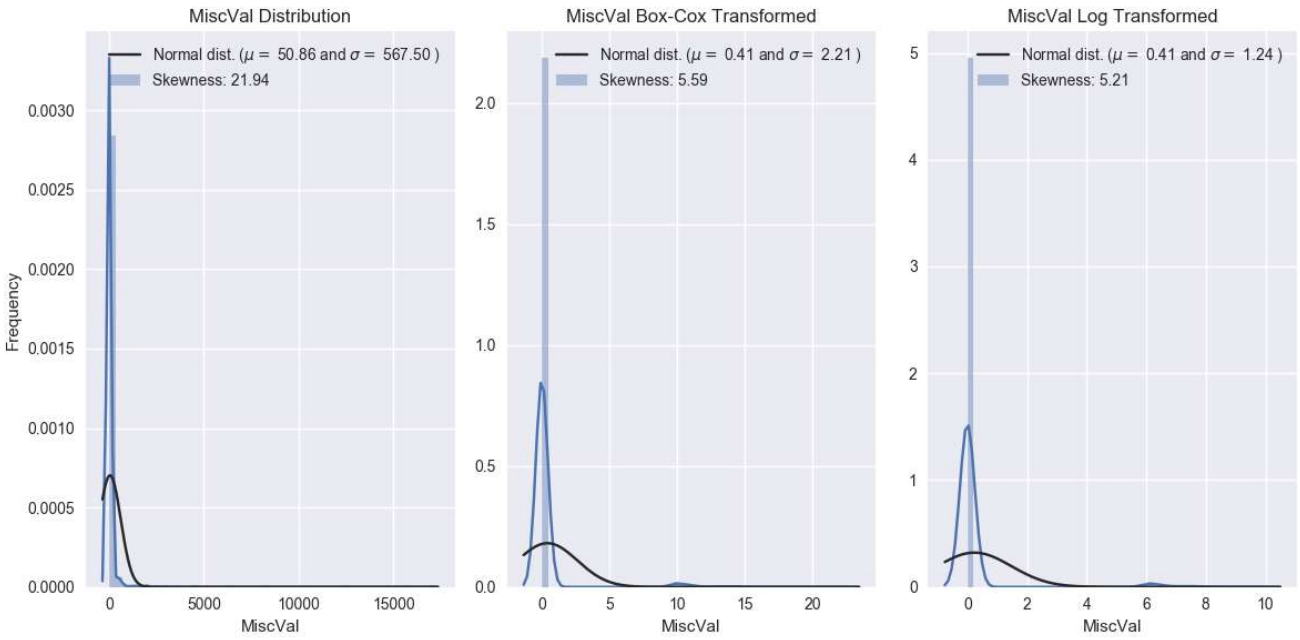
TotalSF was created. TotalSF had higher correlation vs. target than each of its individual components.



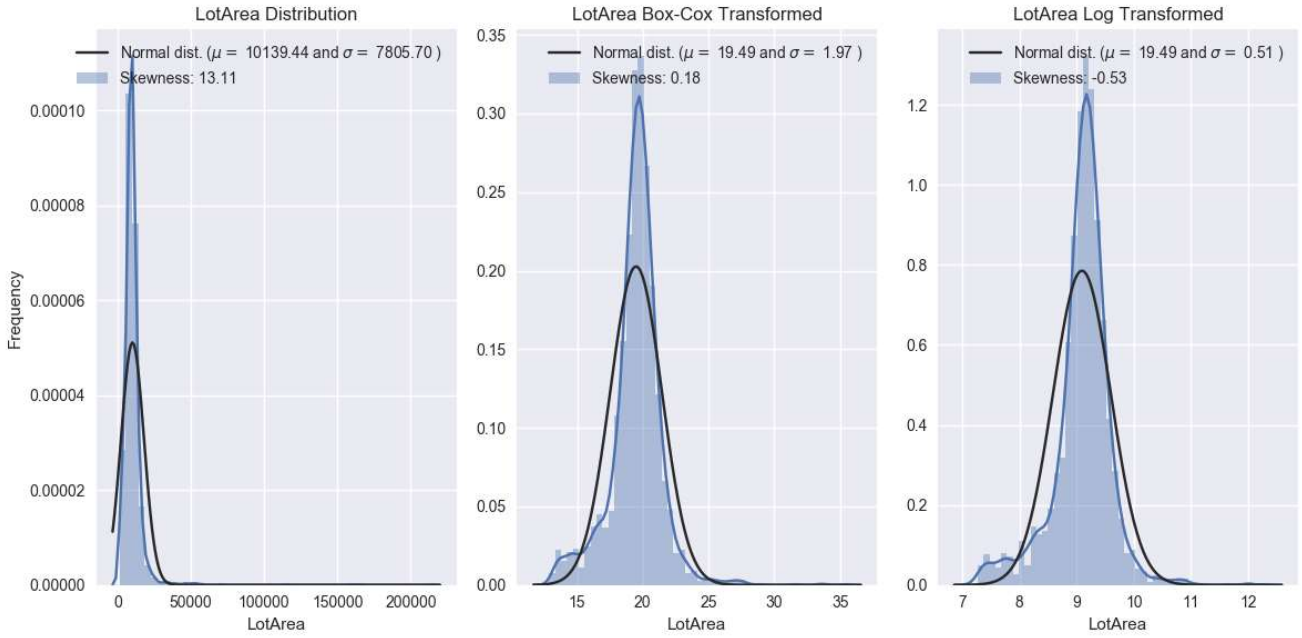
Features Transformed

For each numerical features, I calculated its skew and transformed skews. There were two options for transformations: Box-Cox and Log. Transformations were selected based on skew of distribution:

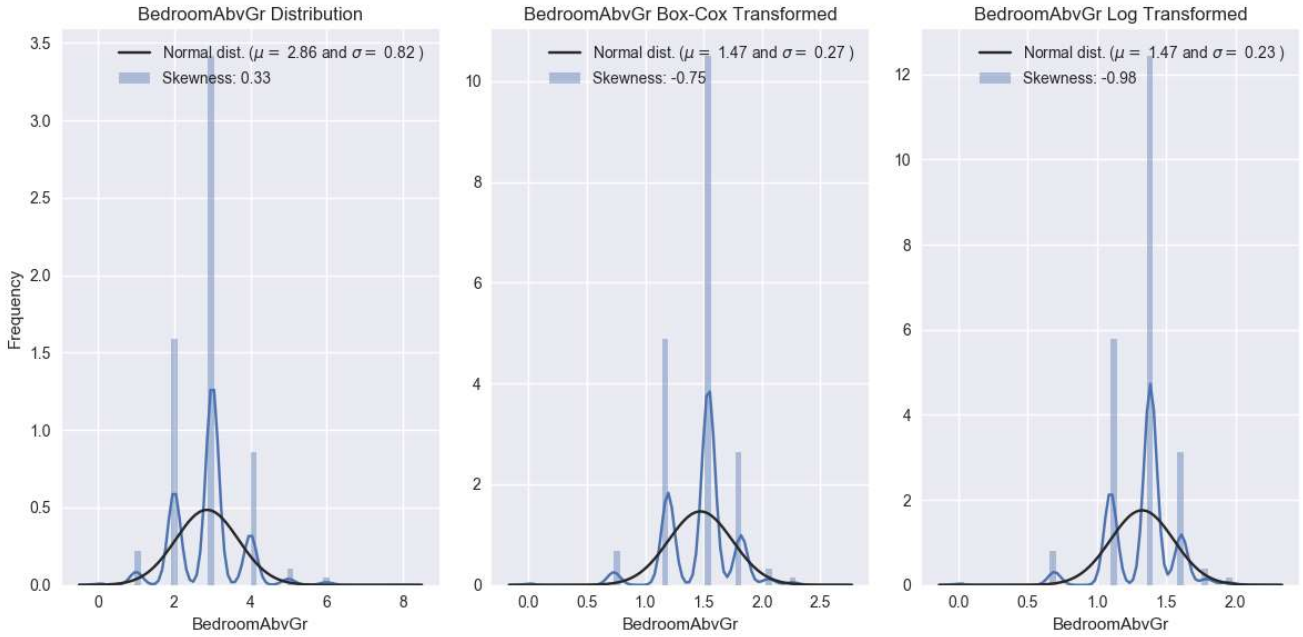
Example of features Log transformed: MiscVal

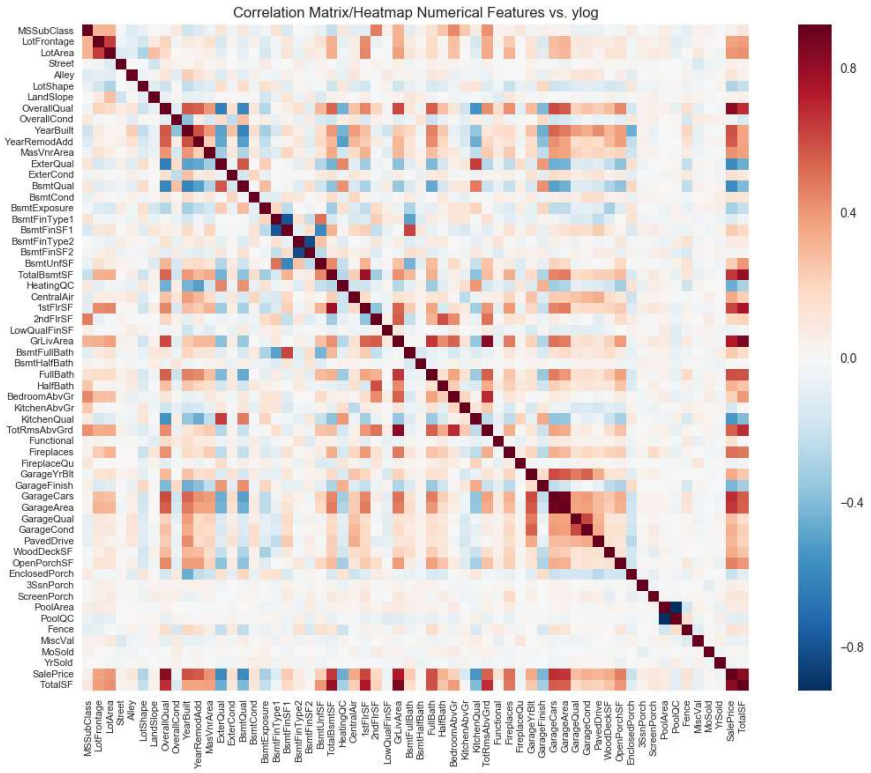


Example of features Box-Cox transformed: LotArea



Example of features not Transformed:

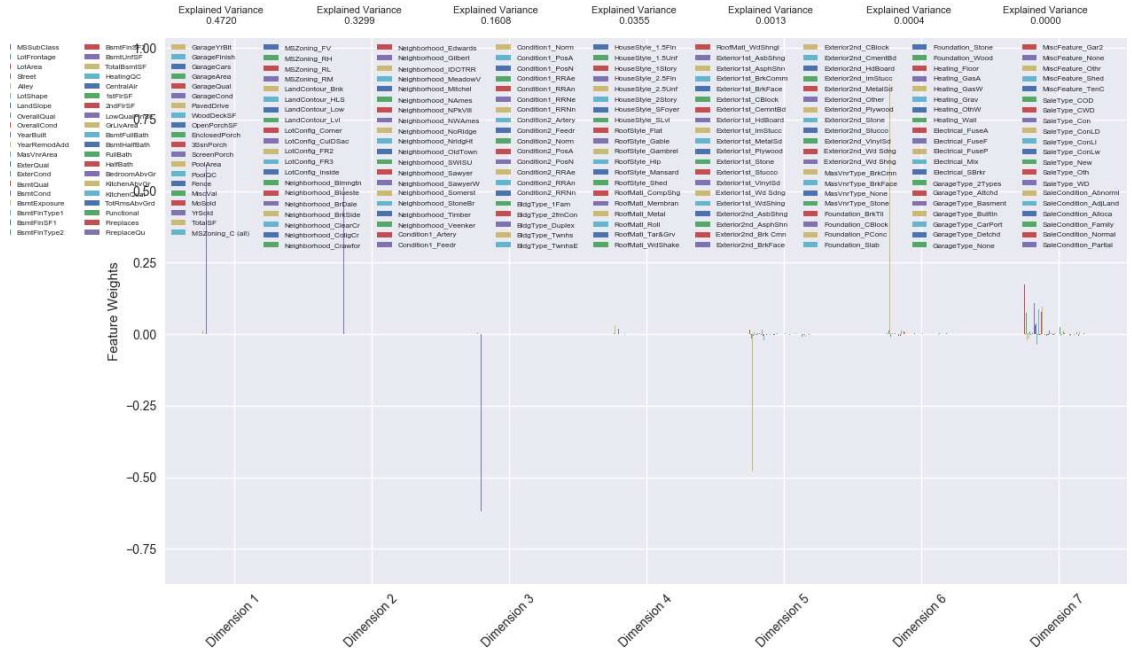




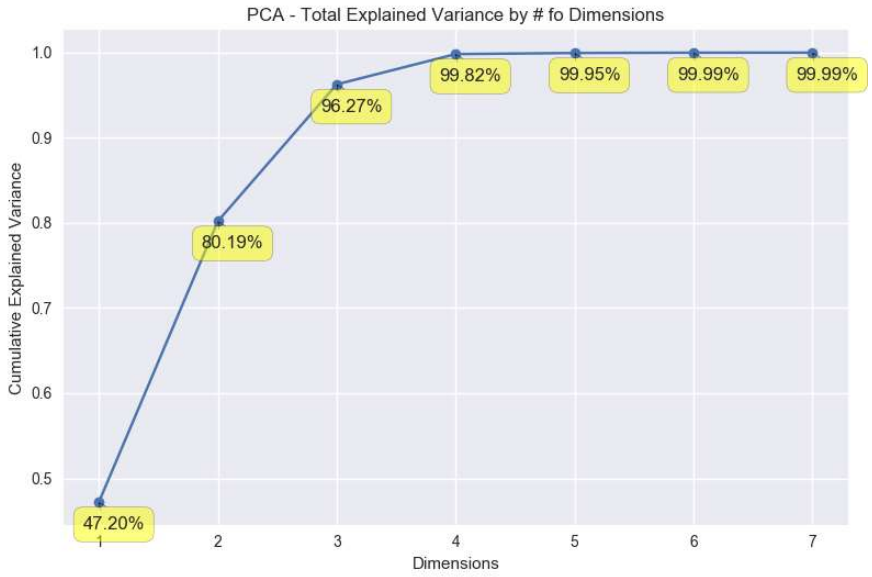
Post feature treatments, I used pandas.corr() and seaborn.heatmap() functions to generate a heatmap to see how features correlated with the target. Top 10 features that correlated to the target were: OverallQual, GrLivArea, GarageCars, GarageArea, FullBath, TotalBsmtSF, 1stFlrSF, YearBuilt, YearRemodAdd, TotalSF

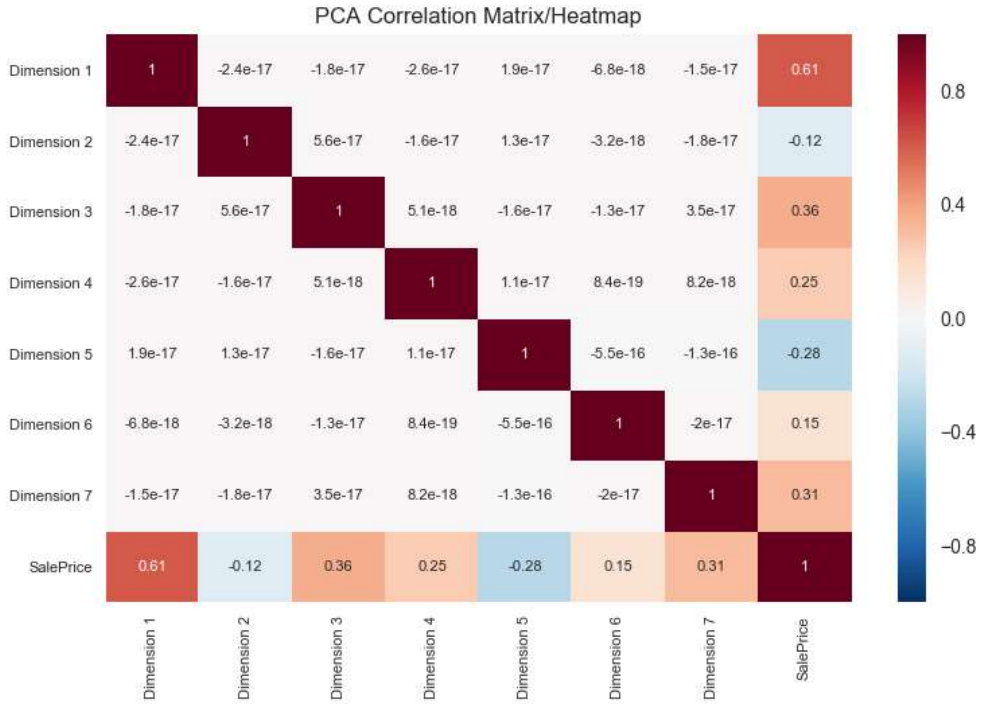
PCA

I ended up with 220 features after feature engineering. Since I would be optimizing several modeling algorithms, some of which were complicated and took a long time to optimize, I felt I should try Principle Component Analysis (PCA). PCA was a statistical procedure that used an orthogonal transformation to convert a set of observations, namely my training and testing data, into a set of values of linearly uncorrelated features, which were called principle components, or ‘dimensions’ in the illustration.



PCA indicated that 6 dimensions were all that PCA needed to explain variance in 220 features from the original training dataset.





The PCA heatmap also indicated some correlation between the target and 7 PCA dimensions. I optimized models on both PCA and non-PCA datasets to compare the effectiveness of PCA.

Algorithms and Techniques

Here is a summary of the algorithms and techniques that are relevant to this exercise. For Lasso, ElasticNet and XGBoost modeling algothrims, I went into details discussion:

- Box-Cox and Log transforms: To improve the distribution of features and target, I transformed using scipy.special.boxcox1p() and np.log1p() functions. For observations that were zero, these functions ensured no error would occur. If features distribution didn't improve, the features would remain un-transformed.
- One-hot: The only way to feed categorical variables into modeling algorithms was to one-hot. sklearn.preprocessing.LabelEncoder() was used.
- PCA: I ended up with 220 features after feature engineering. Since I would be optimizing several modeling algorithms, some of which were complicated and took a long time to optimize, I felt I should try Principle Component Analysis (PCA). PCA was a statistical procedure that used an orthogonal transformation to convert a set of observations, namely my training and testing data, into a set of values of linearly uncorrelated features, which were called principle components, or 'dimensions' in the illustration. PCA indicated that 6 dimensions were all that PCA needed to explain variance in 220 features from the original training dataset. The PCA heatmap also indicated some correlation between the target and 7 PCA dimensions. I optimized models on both PCA and non-PCA datasets to compare the effectiveness of PCA.
- GridSearchCV: For each of the models that I tried, I needed to optimize given different hyperparameters. GridSearch allowed me to implement cross validation and select the optimal hyperparameters at the same time.
- K-Fold Cross Validation: For regression methodologies such as KerasRegressor, it was difficult to implement GridSearchCV. K-Fold was a replacement. It was also much easier to produce/customize visualization with K-Fold instead of GridSearchCV.
- DecisionTreeRegressor: This was a decision tree based regressor (information gain maximization) that had the least hyperparameters and took the least time to optimize. DecisionTreeRegressor was also the benchmark model for this exercise.
- KNeighborsRegressor: KNN was a nearest neighbor based regressor. I wanted to find out how KNN performed compared to Decision Tree.
- RandomForestRegressor: Random Forest was an ensemble method, largely based on the decision tree algorithm. Ensemble methods are highly regarded as models with higher accuracy. I expected Random Forest to perform better than Decision Tree.
- AdaBoostRegressor: Adaptive Boost was also an ensemble method with boosting. Considering its complexity, it should perform better than Random Forest. This also meant potentially lengthier time to optimize.
- GradientBoostingRegressor: Gradient Boost was also an ensemble method with boosting and with even more hyperparameters to adjust. This algorithm performed well with many weak learners. However, I expected lengthy duration to tune this algorithm, especially for inputs with lots of features.

- MLPRegressor: Neural networks were often used for non-linear and complex classification exercises. I wanted to see how neural networks performed in a regression exercise, compare to other more traditional algorithms.
- KerasRegressor: Compared to MLP in sklearn, Keras was more flexible for customization. However, it was also more difficult cross validate. I didn't expect good results from Keras.
- Lasso: Lasso is short for least absolute shrinkage and selector operator. Lasso regression was similar to an ordinary least square regression, where the algorithm was trying to minimize the squared error term. Where the Lasso was different from OLS was the λ term. When λ increased, many of the correlated coefficients would be reduced to zeros. This property was known as feature selection, where Lasso regression only keep features that are uncorrelated and relevant. However, due to feature selection, we may loose information contained in the dataset. Due to Lasso's simplicity, efficiency, and ability to perform feature selection, Lasso could also be a good benchmark for future regression exercises.

  $\min\left(||Y - X\theta||^2\right)$ OLS error term

  $\min\left(||Y - X\theta||_2^2 + \lambda||\theta||_1\right)$ Lasso error term

- ElasticNet: ElasticNet is a combination of Ridge and Lasso algorithms. Ridge regression also contained a λ term, where if λ increased, the error term penalties were also magnified, thereby discouraging overfitting. For ElasticNet, we needed to define Alpha and L1_ratio, where Alpha = λ1 + λ2, and L1_ratio = λ1 / (λ1 + λ2). Note that if L1_ratio = 1, then we have the Lasso algorithm; if L1 = 0, then we have the Ridge algorithm. Therefore, ElasticNet is a combination of give and take between Lasso (feature selection), and Ridge (regularization). Due to ElasticNet's simplicity, efficiency and ability to perform feature selection and regularization, ElasticNet could also bebe a good benchmark for future regression exercise.

  $\min\left(||Y - X\theta||_2^2 + \lambda_1||\theta||_1 + \lambda_2||\theta||_2^2\right)$ ElasticNet error term

  $\min\left(||Y - X(\theta)||_2^2 + \lambda||\theta||_2^2\right)$ Ridge error term

- XGBoost: XGBoost stood for extreme gradient boosting. It was an algorithm based on Gradient Boosting, which combines weak learners into a single strong learner in an iterative fashion. In Gradient Boosting, Gradient descent was use in each iteration to minimize the error function (an additive process). In the case for the Random Forest algorithm, it would generate independent trees with different results, which were aggregated to form the recommendation for the ensemble algorithm. Contrasting Random Forest to Gradient Boosting, instead of having independent trees, trees were added to the previous tree to compliment what the previous tree failed to realize. XGBoost was an algorithm developed to include regularization, which placed emphasis to control for over-fitting. In addition to building a more generalized model, XGBoost also had system optimization and algorithm improvement, which could run more efficiently. As an algorithm that won many kaggle competitions, I expected XGBoost algorithm to perform well.

There were two goals I wanted to accomplish:

1. Compare the effectiveness of engineered features with PCA and without PCA.
2. Compare effectiveness of different regression modeling algorithms. I took in account of duration and RMSE metric to select the ideal solution.

Benchmark

Benchmark was chosen based on ease of feature engineering and regression modeling:

- Feature Engineering: missings imputed, unnecessary features removed, transformed.
- Target Engineering: transformed.
- Regression Modeling: DecisionTreeRegressor, with max_depth = 5

In the most simplistic scenario, we accomplished validation RMSE of 0.1932 and testing RMSE of 0.1725 in less than 0.1 seconds. When compared to the target mean, the benchmark error was 1.61 and 1.43% respectively. Since this was a kaggle competition, it would also be interest to set the bench mark as top 5% of the leaderboard.

III. Methodology

Data Preprocessing

Many ML instructors and articles speak of the importance of making sure the feature distributions are close to normal, and apply PCA to reduce dimensions when necessary. I used this exercise to try to validate these sentiments, and here were the steps that I took:

Target

The target for this exercise was SalePrice. To ensure the distribution was near normal, I selected log transformation based on skew of data distributions. The probability plot also indicated log transformation was the best choice.

Outliers

Scrubbing through feature distributions, I found that only GrLivArea have two data points that were located outside of the general trend and the general cluster of the feature vs. target. I removed these two data points, where GrLivArea > 4000 and SalePrice < 200000, as these two data points would likely negatively impact the modeling results.

Missing Data

There were 36 features that contained missing data. Out of those, 4 features with missings exceeded 80%. Several treatments were implemented for the missing values:

- Missing replaced by None – for these categorical features, the missing values should be replaced by ‘None’. Namely: PoolQC, MiscFeature, Alley, Fence, FireplaceQu, GarageType,

GarageFinish, GarageQual, GarageCond, BsmtQual, BsmtCond, BsmtExposure, BsmtFinType1, BsmtFinType2, MasVnrType.

- Missing replaced by zero – for these numerical features, missing values should be replaced by 0. Namely: GarageYrBlt, GarageArea, GarageCars, BsmtFinSF1, BsmtFinSF2, BsmtUnfSF, TotalBsmtSF, BsmtFullBath, BsmtHalfBath, MasVnrArea
- Missing replaced by mode – mode were reasonable replacement value of these features. Namely: MSZoning, Electrical, KitchenQual, Exterior1st, Exterior2nd, SaleType
- Missing replaced by specific value – the missings in the Functional feature should take on the value 'Typ', indicating typical.
- Missing replaced by median – these features were likely dependent on other features as the feature LotFrontage for the same neighborhood should be similar; therefore, missing LotFrontage took on median values of those in the same neighborhoods.

Delete Feature

Meaningless features, such as Id, which indicated the record number in the dataset, was removed. Features such as Utilities was also removed due lack of variance.

Feature Types Transformed

Some numerical features were transformed into categorical features because they were more reasonable as categorical features. Namely: MSSubClass, OverallCond, YrSold, MoSold

One-hot Features

Categorical features that might correlate with the targets were treated with one-hot encoding using sklearn.preprocessing.LabelEncoder(). Namely: FireplaceQu, BsmtQual, BsmtCond, GarageQual, GarageCond, ExterQual, ExterCond, HeatingQC, PoolQC, KitchenQual, BsmtFinType1, BsmtFinType2, Functional, Fence, BsmtExposure, GarageFinish, LandSlope, LotShape, PavedDrive, Street, Alley, CentralAir, MSSubClass, OverallCond, YrSold, MoSold

New Features created

TotalSF was created from adding three other features, namely TotalBsmtSF, 1stFlrSF and 2ndFlrSF. The new feature had higher correlation with the target than each of its individual components.

Features Transformed

For each numerical features, I calculated its skew and transformed skews. There were two options for transformation: Box-Cox and Log. Transformations were selected based on resulting skew:

- Log transformed features: MiscVal, PoolArea, LowQualFinSF, 3SsnPorch, LandSlope, KitchenAbvGr, BsmtFinSF2, EnclosedPorch, ScreenPorch, BsmtHalfBath, MasVnrArea, OpenPorchSF, WoodDeckSF, 1stFlrSF, GrLivArea, 2ndFlrSF, TotRmsAbvGrd, Fireplaces, HalfBath, BsmtFullBath, HeatingQC

- Box-Cox transformed features: LotArea, LotFrontage, TotalSF, BsmtFinSF1

Post feature treatments, I also used pandas.corr() and seaborn.heatmap() functions to generate a heatmap to see how features correlate with the target. Top 10 features that correlates to the target were: OverallQual, GrLivArea, GarageCars, GarageArea, FullBath, TotalBsmtSF, 1stFlrSF, YearBuilt, YearRemodAdd, and TotalSF.

PCA

With 220 features after feature engineering, some modeling algorithms might be very computationally expensive to optimize. In order to reduce run time of models and dimensionality of the dataset, I used PCA to boil 220 features down to 6 dimensions. 6 dimensions were all it required to explain all the variance in the original training dataset.

Implementation

There were a total of 2 datasets and 10 modeling algorithms, resulting in 20 opportunities to optimize the model.

I created one file (Feature_Engineering.py) that would take care of feature engineering and export of data, and two more files (Model Optimization GridsearchCV.py and Model Optimization Kfold.py) to explore the possibilities of modeling algorithms and dataset combinations.

In the modeling exploration phase of the exercise, I used K-Fold to get a sense of how each hyperparameter affected the metric. It was also much easier to access modeling results and build visualizations to illustrate the relationship between hyperparameters and metrics with the K-Fold code.

After receiving some intuition with K-Fold, I used GridSearchCV to search for the hyperparameter combinations that would optimize each modeling algorithms. All hyperparameters, duration, and train and test errors were recorded. All of these outputs were considered in selecting an optimal modeling algorithm.

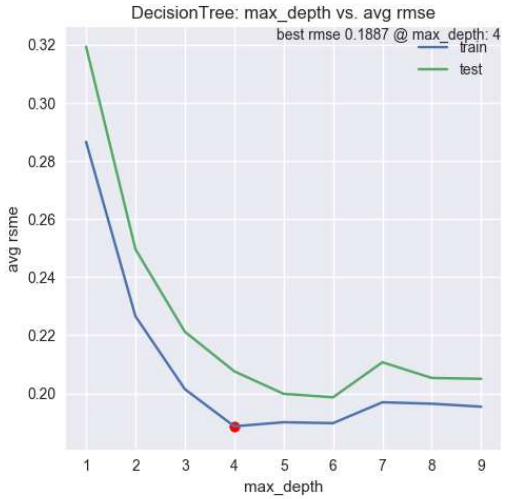
Refinement

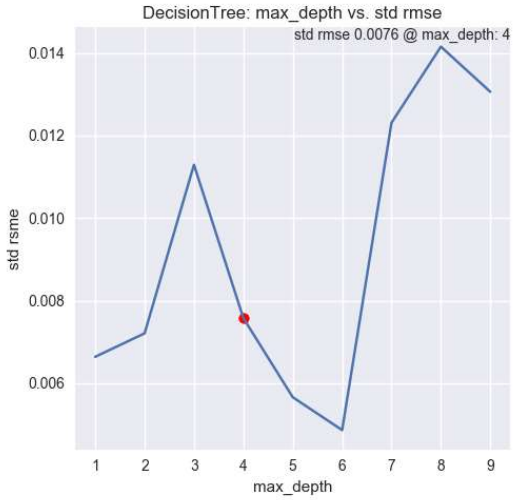
In the modeling exploration phase of the exercise, I used K-Fold to get a sense of how each hyperparameter affected the metric for each modeling algorithm. With this process, I could also gauge the ranges I should implement during the GridSearchCV process.

For each model and exploration of the hyperparameters, I plotted train and test RMSE vs. hyperparameter, and RMSE standard deviation vs. hyperparameters. The optimal hyperparameter, based on avg train RMSE, was also indicated on the plots.

DecisionTreeRegressor (benchmark)

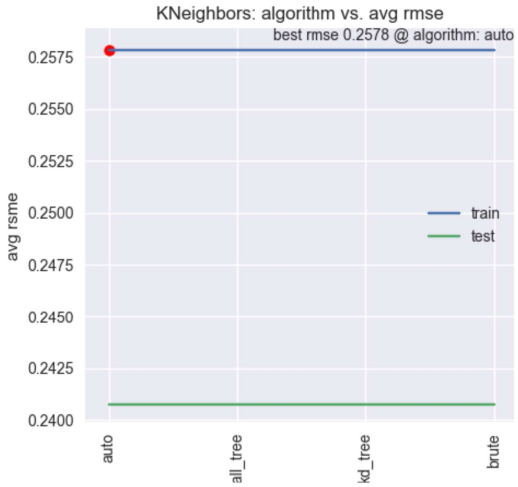
One hyperparameter to optimize: as max_depth exceeded 5, there were some overfitting. I would expect the optimal max_depth to be around 4-5.

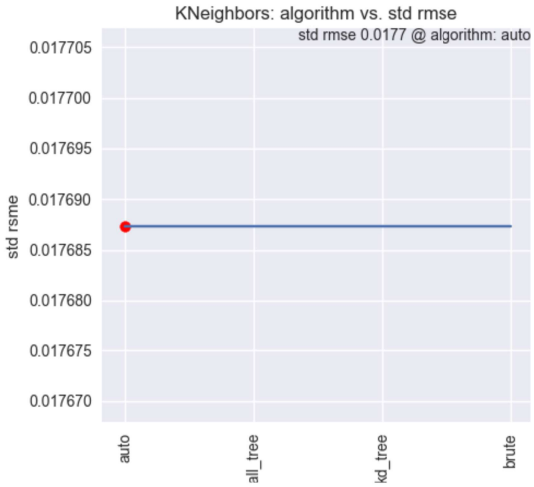


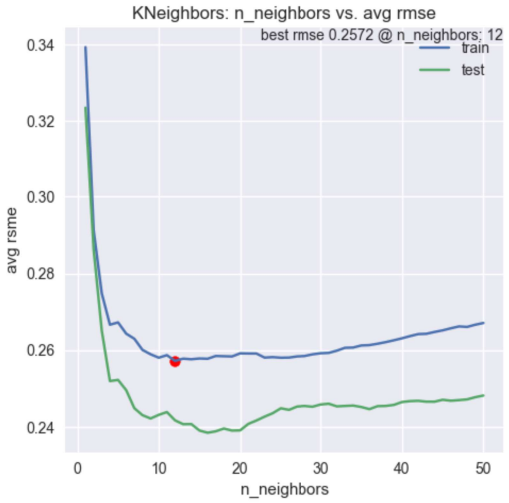


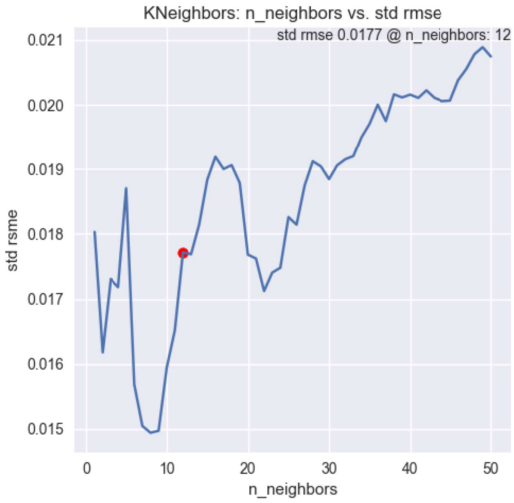
KNeighborsRegressor:

Two hyperparameters to optimize: algorithms didn't seem to make a difference, while n_neighbors around 12 ~ 15 seemed to yield the best result.



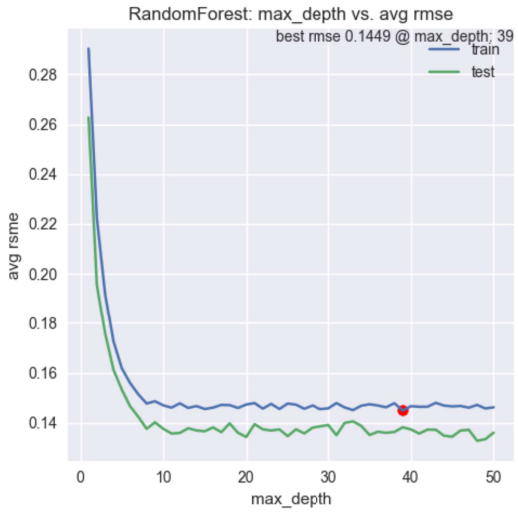


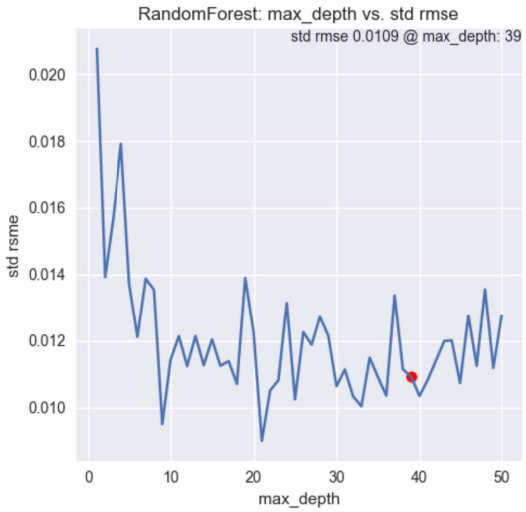


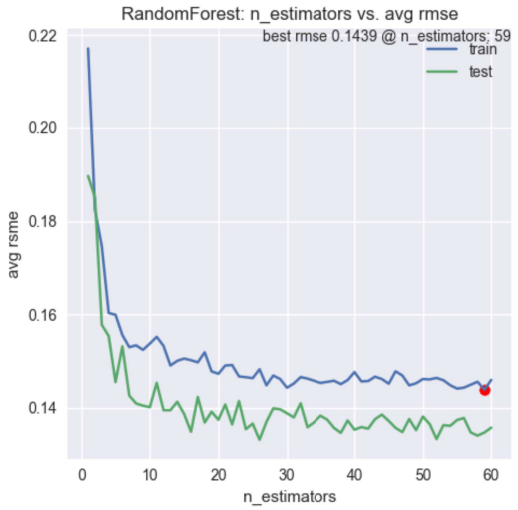


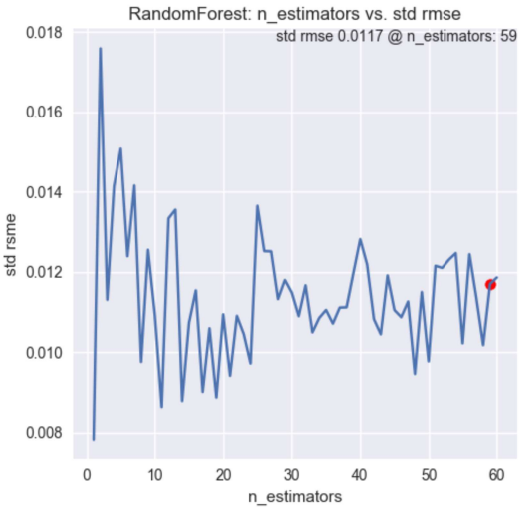
RandomForestRegressor:

Two hyperparameters to optimize for Random Forest: RSME seemed to be stable for max_depth > 20, while n_estimators might need to take on values that are greater than 55.



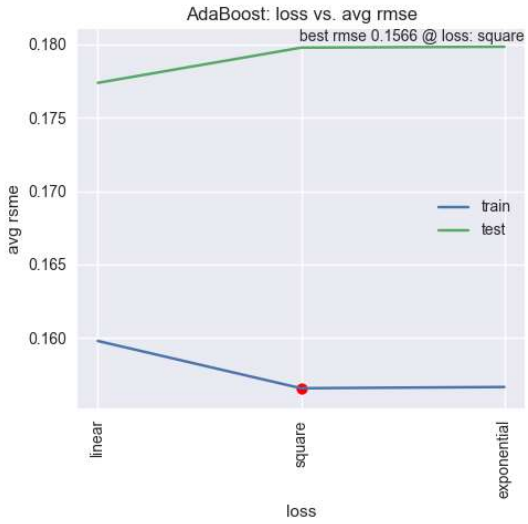


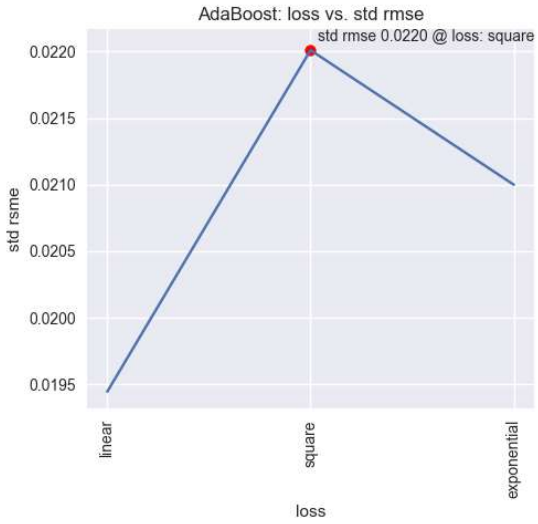


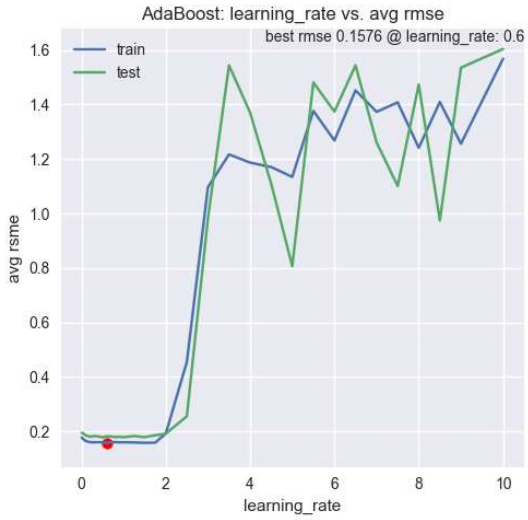


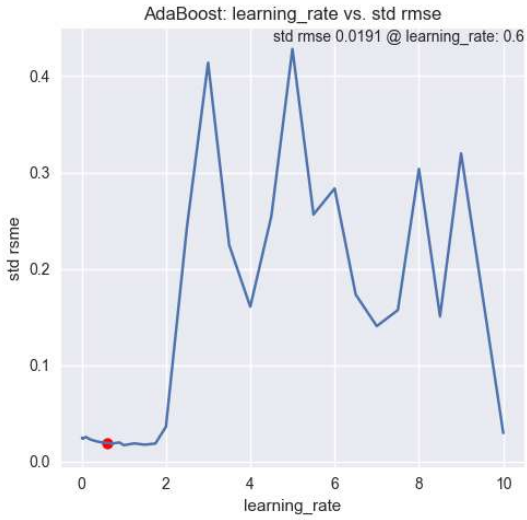
AdaBoostRegressor:

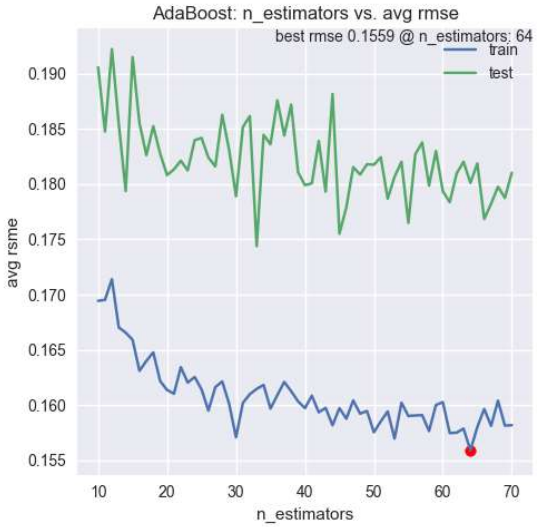
Three hyperparameters to optimize: loss could take on either square or exponential values, learning_rate needed to be less than 2, and n_estimators might need to take on values > 60.

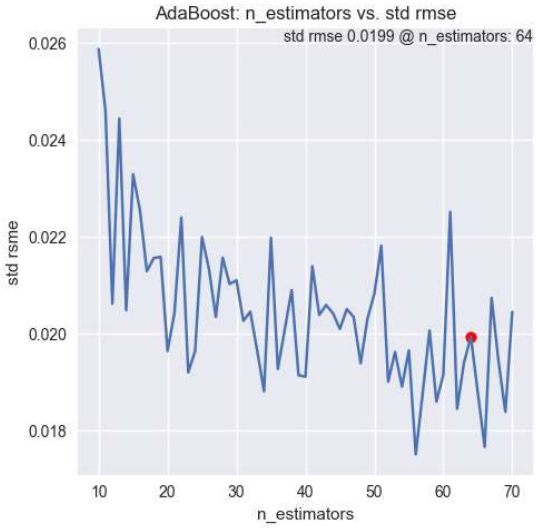






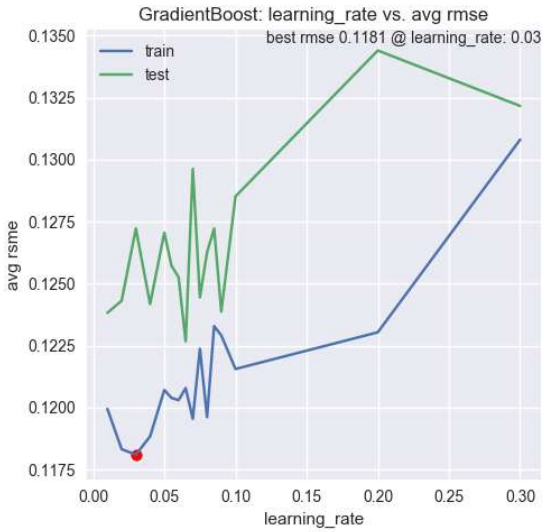


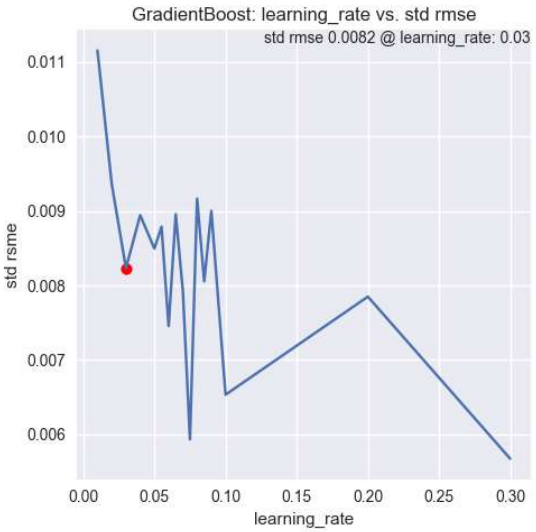


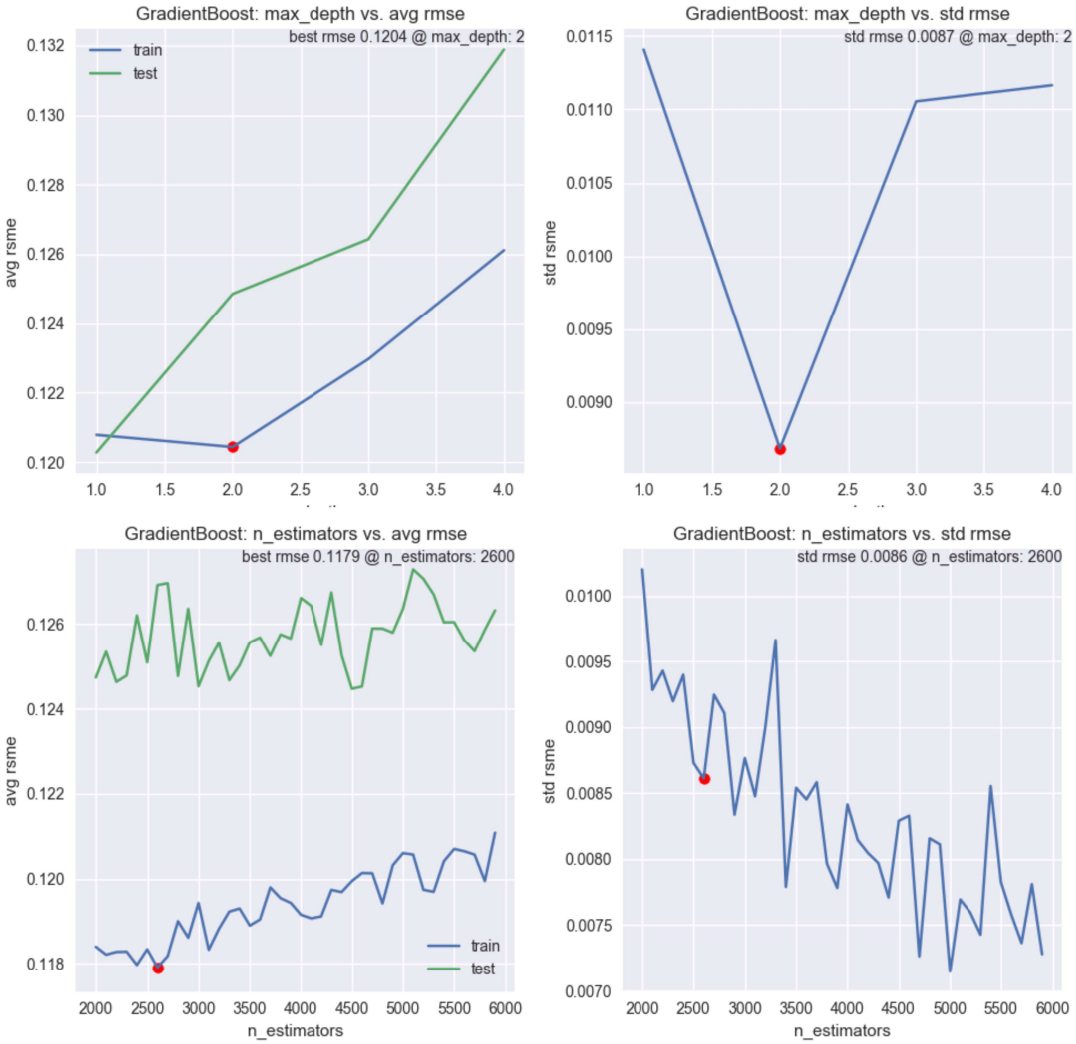


GradientBoostingRegressor:

Three hyperparameters to optimize: learning_rate needed to be less than 0.05, max_depth needed to be around 2, and n_estimators needs to be < 3000.

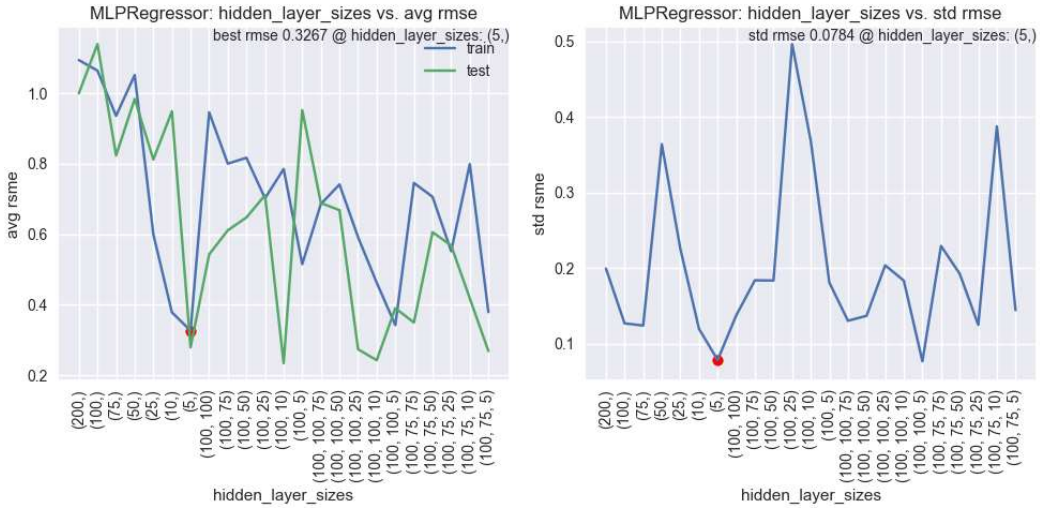


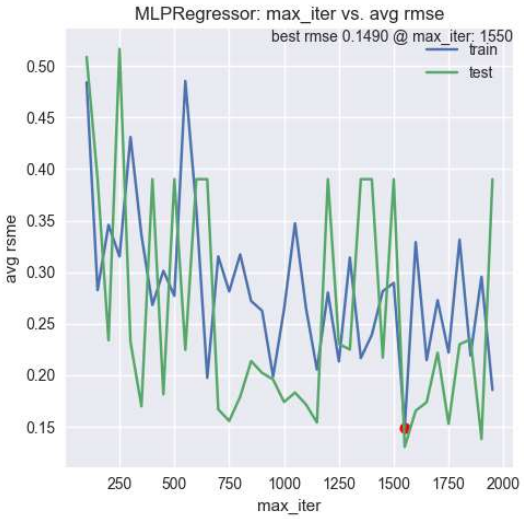


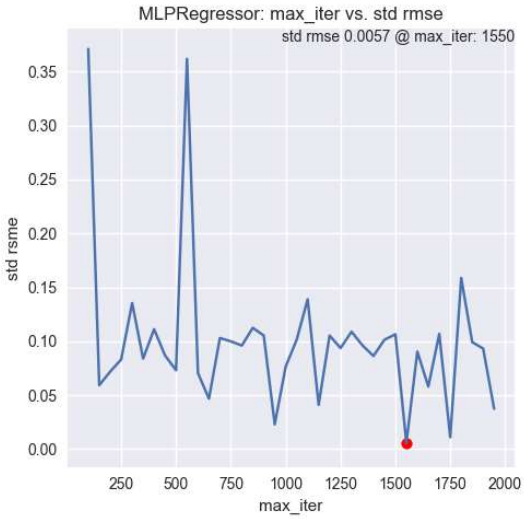


MLPRegressor:

Two hyperparameters to optimize: hidden_layer_sizes seemed optimal at (5,), and max_iter needed to be around 1500.

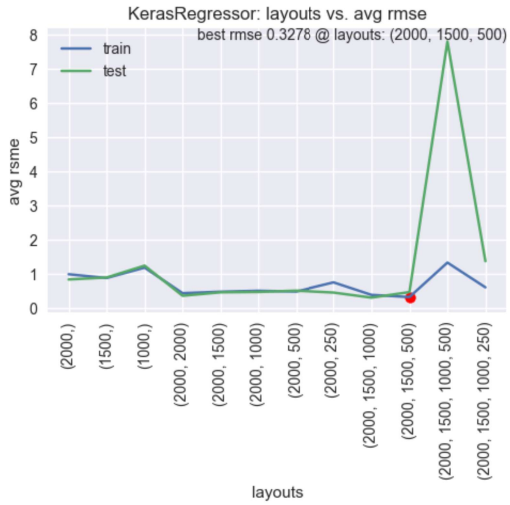


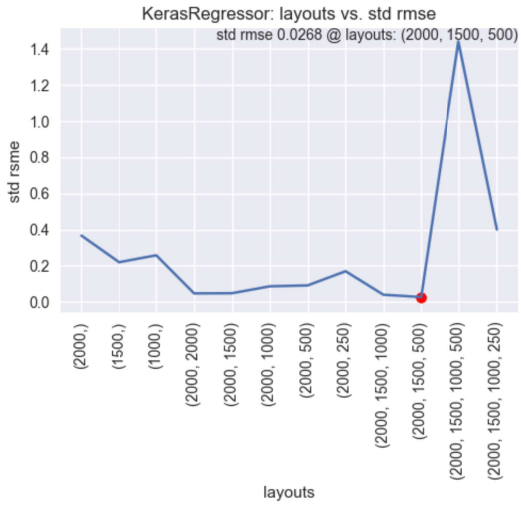


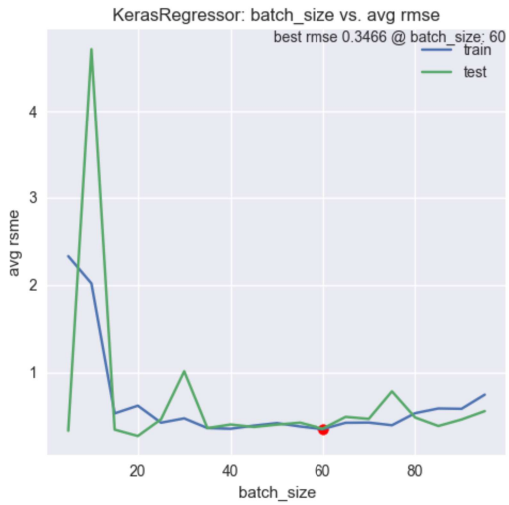


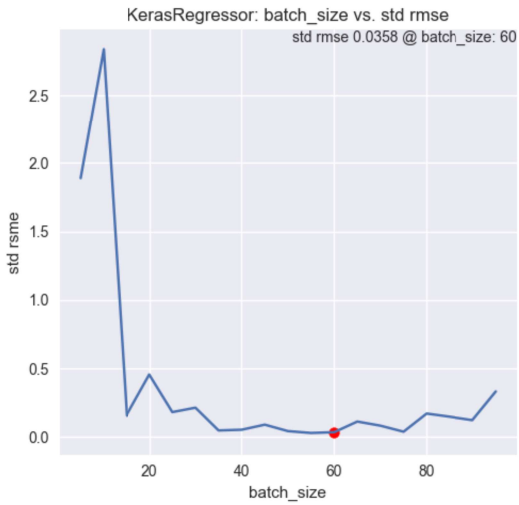
KerasRegressor:

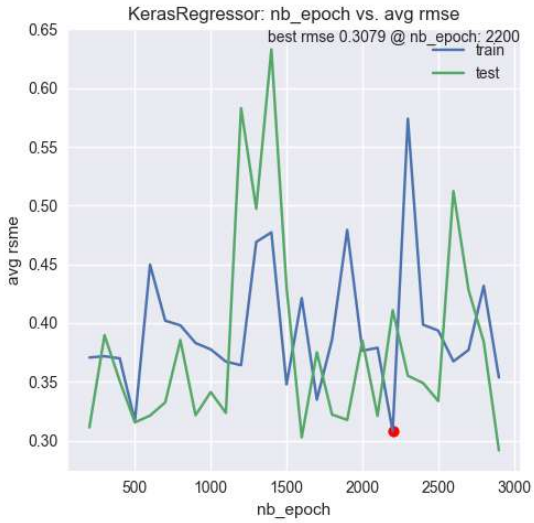
Three hyperparameters to optimize: layouts (self made methods) optimal at (2000, 1500, 500), batch_size needed to be around 60, and nb_epochs didn't have a clear trend.

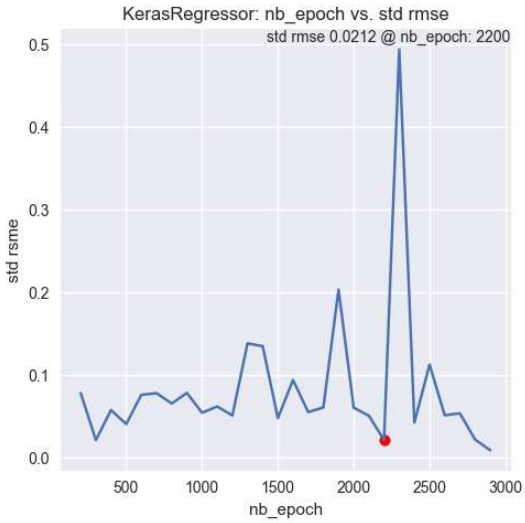






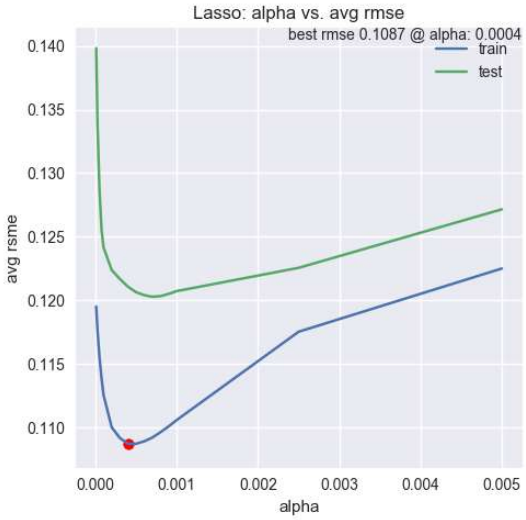


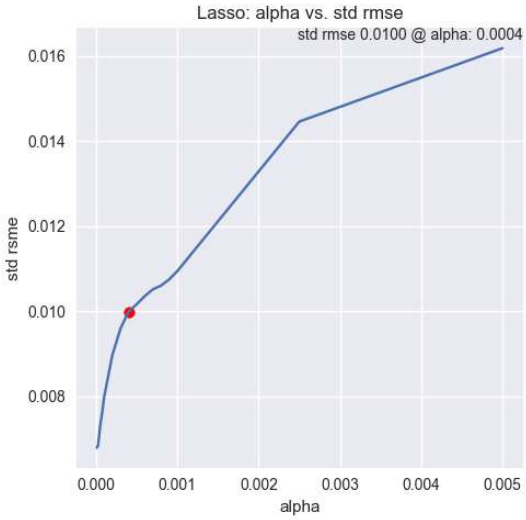




Lasso:

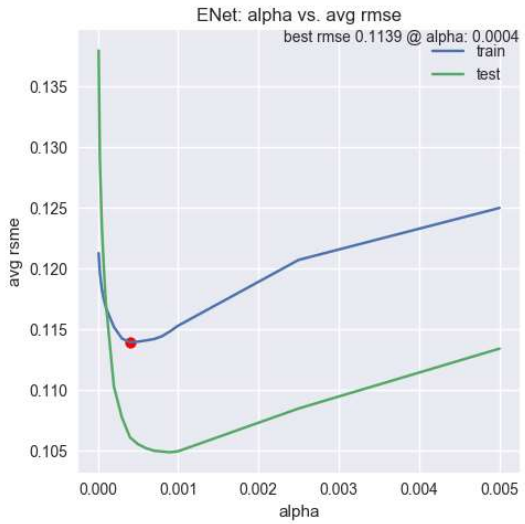
One hyperparameter to optmize: alpha between 0.0002 and 0.0005 should perform well

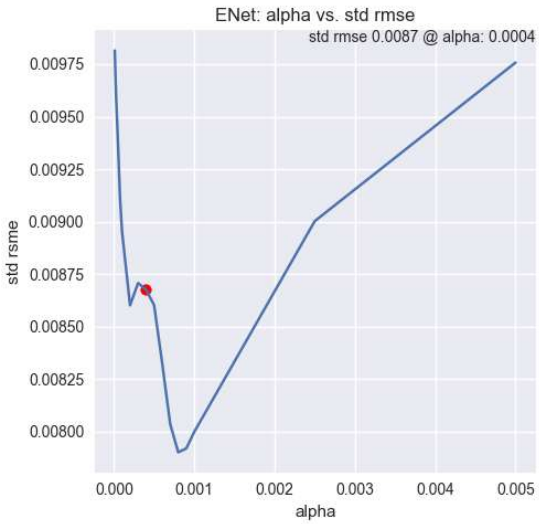


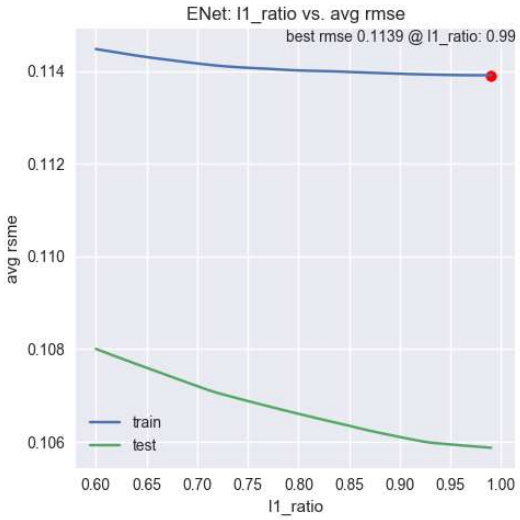


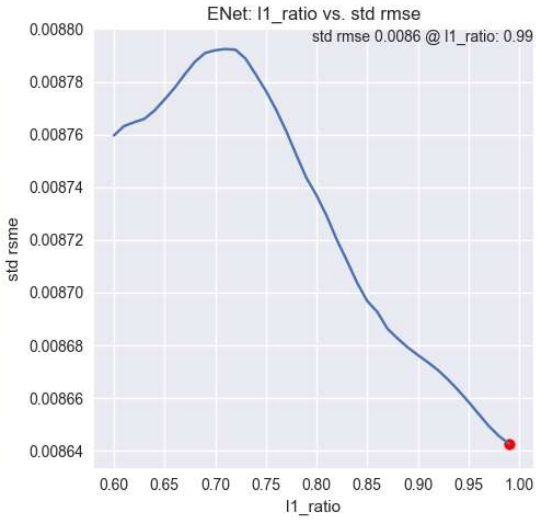
ElasticNet:

Two hyperparameters to optimize: alpha between 0.0002 and 0.0005 should perform well, while l1_ratio needed to be greater than 0.9



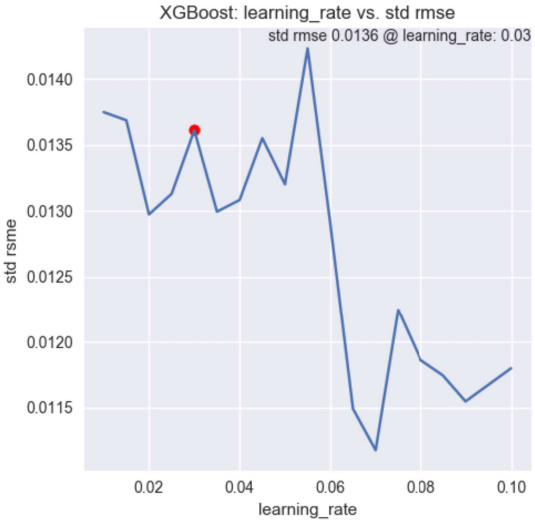


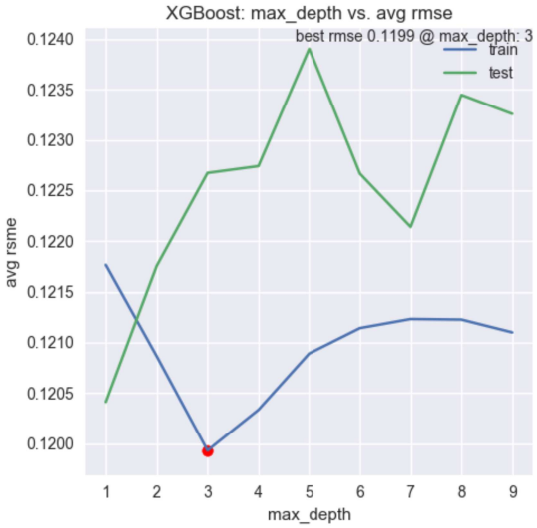


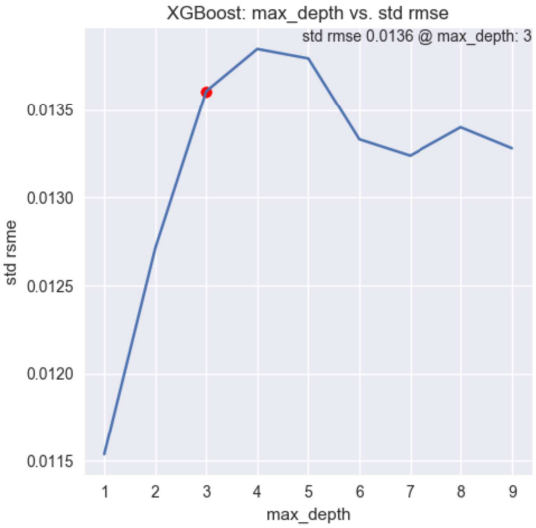


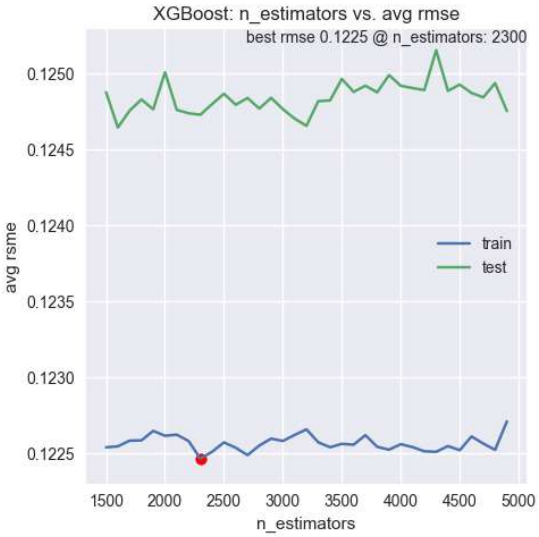
<u>XGBoost</u>:

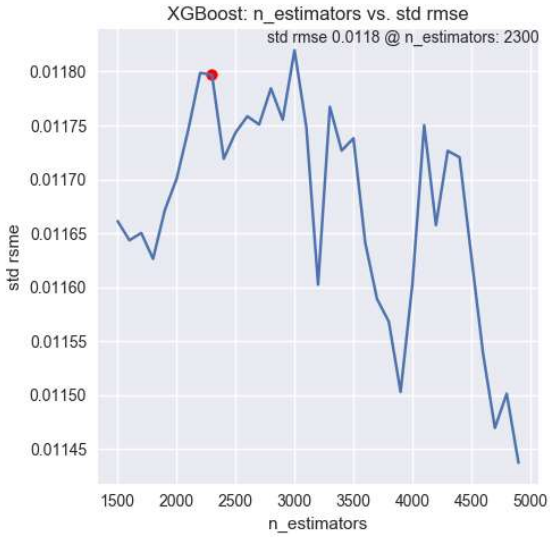
Three hyperparamers to optimize: learning rate seemed to work best between 0.03 and 0.04; max_depth needed to be 3; RMSE were quite consistent for a wide range of of n_estimators.









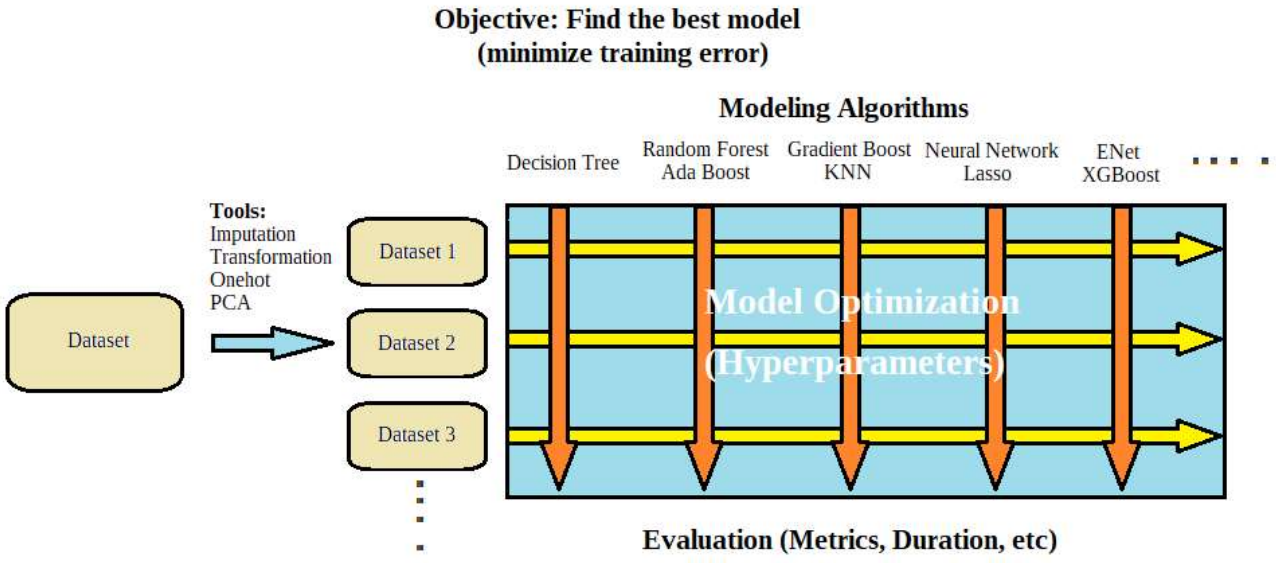


It was difficult and time consuming to implement KerasRegressor and MLPRegressor with GridSearchCV. Even though I could integrate GridSearchCV to optimize Keras' hyperparameters, KerasRegressor did not inherently attempt to define the most optimal structure, as a Random Forest or XGBoost algorithm would. Ideally, I wanted to specify max depth, max neuron for all layers, dropout or not, and choices for activation functions. However, that wasn't possible and all those 'hyperparameters' were the art and science of neural networks. For each neural network structure that I tried, I ended up writing methods to represent them in the cross validation methodology, which meant the coding efficiency suffered. Duration was also a concern for KerasRegressor and MLPRegressor algorithms, even with the PCA dataset. After optimization, results were much worse than algorithms that took fraction of their time. All in all, Neural Network regressors were not ideal in predicting home prices.

IV. Results

Model Evaluation and Validation

As the exercise progressed through my matrix of datasets and modeling algorithms, certain algorithms stood out and consistently outperformed the benchmark and their competitions:



K-Folds methodology winners: Lasso ElasticNet, and XGBoost

GridSearchCV methodology winners: Lasso, ElasticNet, and XGBoost

Featureset	Regressor Type	GridSearch Methodology				K-Fold Methodology (Reference)			
		Optimized Hyperparameter	Time Elapsed	Optimized Train/Validation Score	Optimized Test Score	Optimized Hyperparameter	Time Elapsed	Optimized Train/Validation Score	Optimized Test Score
no PCA	Decision Tree (Benchmark)	max_depth ~ 5	< 0.1 Second	0.1932 (1.61%)	0.1725 (1.43%)	max_depth ~ 6	< 0.1 Seconds	0.1900 (1.58%)	0.1746 (1.45%)
	Random Forest	max_depth ~ 22 n_estimators ~ 28 Bootstrap = True	< 5 Seconds	0.1390 (1.16%)	0.1223 (1.02%)	max_depth ~ 39 n_estimators ~ 59	< 5 Seconds	0.1439 (1.20%)	0.1347 (1.12%)
	AdaBoost	loss = 'exponential' learning_rate ~ 5.5 n_estimators ~ 54	< 5 Seconds	0.1807 (1.50%)	0.1865 (1.55%)	loss ~ 'exponential' learning_rate ~ 3.5 n_estimators ~ 60	< 5 Seconds	0.1641 (1.36%)	0.1602 (1.33%)
	Gradient Boost	learning_rate ~ 0.12 max_depth ~ 4 n_estimators ~ 300	~ 20 Seconds	0.1233 (1.03%)	0.1162 (0.97%)	max_depth ~ 2 learning_rate ~ 0.03 n_estimators ~ 4000	~70 Seconds	0.1219 (1.01%)	0.1073 (0.89%)
	K Nearest Neighbors	algorithms = 'auto' n_neighbors ~ 20	< 1 Second	0.2474 (2.06%)	0.2580 (2.15%)	algorithm = 'auto' n_neighbors ~ 13	< 1 Seconds	0.2602 (2.16%)	0.2239 (1.86%)
	MLP	hidden_layer_sizes = (3000, 2000) max_iter ~ 2200	~300 Seconds	0.5245 (4.36%)	0.5186 (4.31%)	hidden_layer_sizes = (5,) max_iter ~ 1550	< 5 Seconds	0.1490 (1.24%)	0.1306 (1.08%)
	Keras	hidden_layer_sizes = (2000,) max_iter ~ 2200 Optimizer = 'rmsprop' batch_size ~ 25 epochs ~ 100	~100 Seconds	0.4704 (3.91%)	7949.4912 (66011.74%)	Layout = (2000, 1500, 1000) batch_size ~ 40 nb_epoch ~ 2800	~ 200 Seconds	0.3051 (2.54%)	0.3533 (2.94%)
	Lasso	Alpha ~ 0.0006	< 1 Second	0.1118 (0.93%)	0.1017 (0.84%)	Alpha ~ 0.0004	< 1 Seconds	0.1119 (0.93%)	0.1097 (0.91%)
	Enet	alpha ~ 0.013 l1_ratio ~ 1e-07	< 1 Second	0.1115 (0.93%)	0.1044 (0.87%)	Alpha ~ 0.0005 l1_ratio ~ 0.99	< 1 Seconds	0.1139 (0.95%)	0.1059 (0.88%)
	XGBoost	learning_rate ~ 0.08 max_depth ~ 4 n_estimators ~ 300 reg_alpha ~ 0.002	< 5 Seconds	0.1190 (0.99%)	0.1087 (0.90%)	n_estimators ~ 2300 learning_rate ~ 0.03 max_depth ~ 3	~ 20 Seconds	0.1199 (1.00%)	0.1227 (1.02%)
PCA	Decision Tree	max_depth ~ 4	< 1 Second	0.2565 (2.13%)	0.2657 (2.21%)				
	Random Forest	max_depth ~ 29 n_estimators ~ 19 Bootstrap = True	< 1 Second	0.1953 (1.62%)	0.1841 (1.53%)				
	AdaBoost	loss = 'exponential' learning_rate ~ 5.5 n_estimators ~ 56	< 1 Second	0.2217 (1.84%)	0.2184 (1.82%)				
	Gradient Boost	learning_rate ~ 0.12 max_depth ~ 4 n_estimators ~ 300	< 1 Second	0.1926 (1.60%)	0.1805 (1.50%)				
	K Nearest Neighbors	algorithms = 'auto' n_neighbors ~ 20	< 1 Second	0.2473 (2.06%)	0.2581 (2.15%)				
	MLP	hidden_layer_sizes = (5, 5) max_iter ~ 3000	< 5 Seconds	2.0904 (17.39%)	0.9042 (7.52%)				
	Keras	hidden_layer_sizes = (2000,) max_iter ~ 2200 optimizer = 'adam' batch_size ~ 25 Epochs ~ 1000	~40 Seconds	0.5168 (4.30%)	0.4066 (3.38%)				
	Lasso	Alpha ~ 0.0005	< 1 Second	0.1857 (1.55%)	0.1801 (1.50%)				
	Enet	alpha ~ 1.6e-05 l1_ratio ~ 0.09	< 1 Second	0.1857 (1.55%)	0.1800 (1.49%)				
	XGBoost	learning_rate ~ 0.08 max_depth ~ 4 n_estimators ~ 300 reg_alpha ~ 0.002	< 1 Second	0.1917 (1.59%)	0.1847 (1.54%)				

I would consider Lasso as the top performing algorithm of the exercise. With engineered features without PCA, Lasso achieved the best test metric, which was a 31% improvement over the Decision Tree benchmark. Lasso was simple to optimize and took < 1 second on average to fit to the data, making the selection not only effective, but also very efficient.

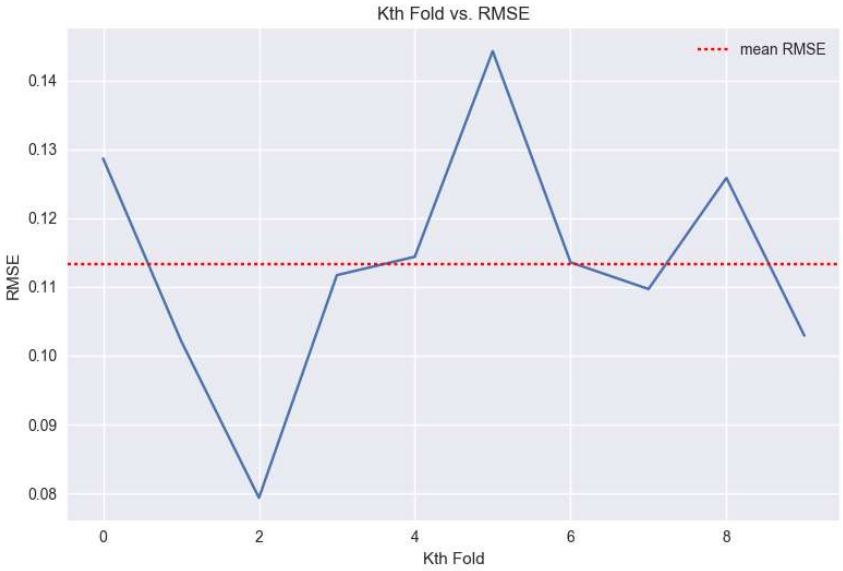
Justification

From the model algorithm and dataset grid, the winning model and dataset combination was no PCA dataset and Lasso. The optimized hyperparamers was alpha (0.0006), which protected against overfitting. Here were Lasso's performance metrics:

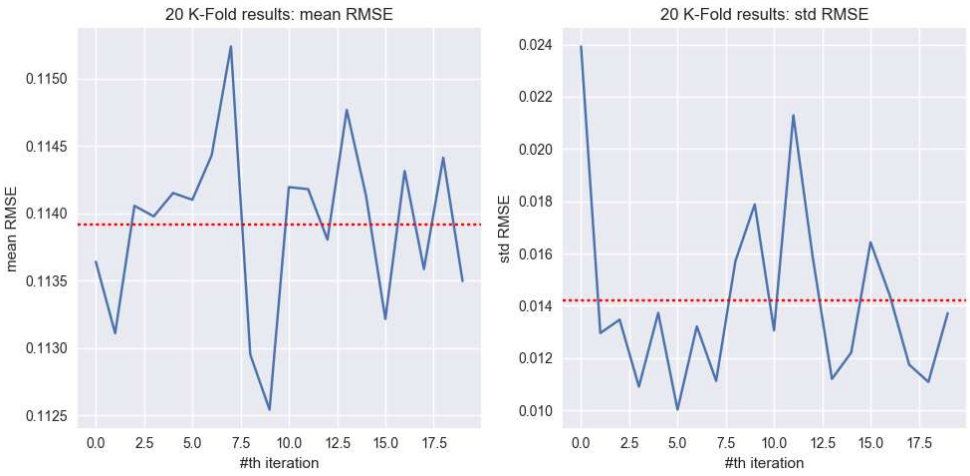
- Training RSME is 0.1104 (0.92% of target)
- Test RSME is 0.1039 (0.86% of target)
- Model train time < 1 second

I also verified the robustness of the Lasso Model prior to making the decision:

RMSE for each fold during K-Fold exercise – resulting RMSE were quite consistent with average RMSE:

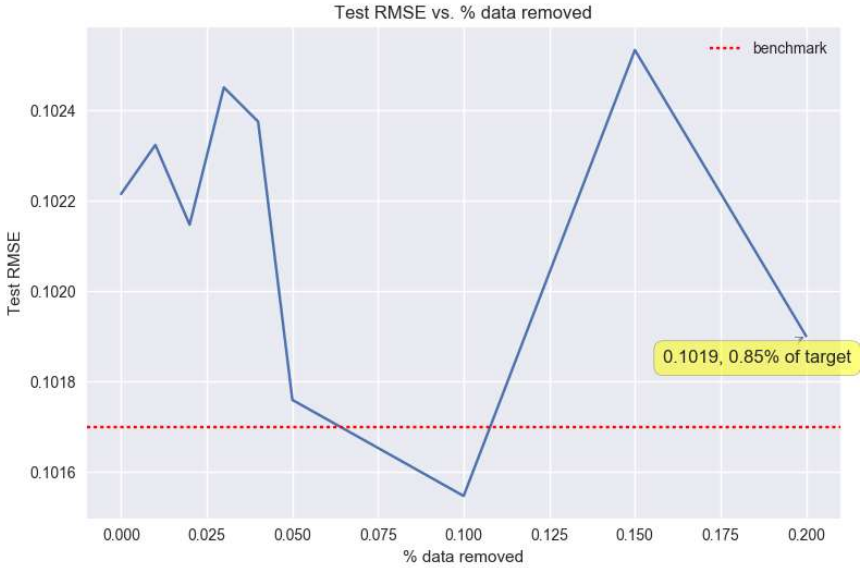


Cross Validation with different random states – after performing 20 K-Folds, the average RMSE and and RMSE std were quite consistent, except for a few outliers.

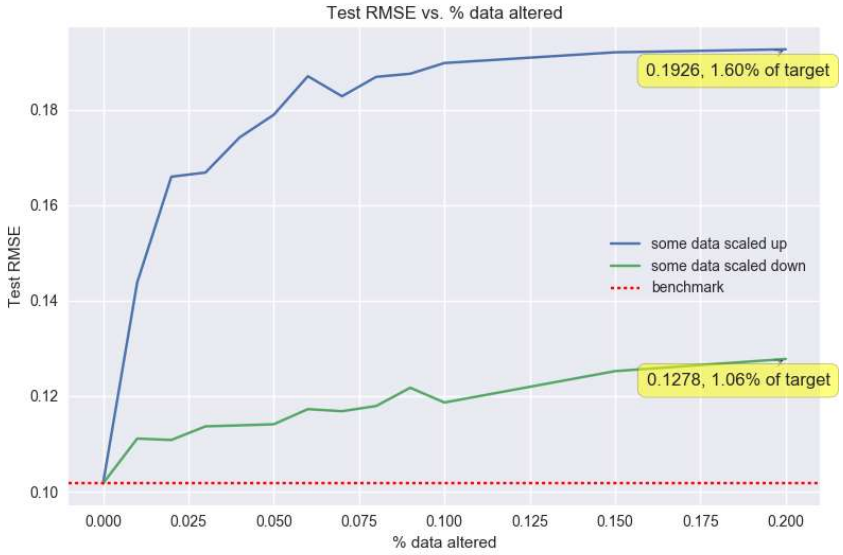


Adjustments to the modeling dataset:

% training data removed – with some data removed (up to 20%), we are still seeing the Lasso Regressor performing reasonably well, much better than benchmark.



% training data scaled up by 10x or scaled down by 10x – I was quite surprised by the results here, as some data (up to 20%) were scaled up or down, Lasso Regressor still performed better than benchmark. Note that the scaled down dataset had less impact to modeling result than the scaled up dataset.



Overall, Lasso outperformed the Decision Tree (benchmark) by 42% on the training dataset and 41% on the testing dataset. With the cross validation methodology, I was confident that Lasso consistently outperformed the benchmark and generalized well.

V. Conclusion

Reflecting back on my efforts to perform regression and predict housing prices, I realized that this was more about art and science of modeling than trying to minimize the metrics. Without instruction or guidance, I had to search for and understand a dataset, perform feature engineering, select suitable models, optimize modeling algorithms, and compare results from combinations of datasets and modeling algorithms. This exercise would set a precedence for future modeling exercises, where I would follow similar methodologies to make informed decisions on model selection.

Free-Form Visualization

Residual plot for the Lasso Regressor – ideally, the model residual plots would have correlations of zeros. There were still some correlations between residuals and training and testing targets, indicating that I could still do better in finding a model that would better predict SalePrice.

Reflection

To reflect back on the project by project milestones:

1. Explore data
    - Interpreting and understanding the features and their values were critical.
    - Visualizations were essential in becoming domain experts.
2. Clean data
    - Using intuitions to comb through the features, and to figure out which features were relevant, and what the best ways were to impute missing values.
3. Feature treatments
    - PCA was not always helpful – I optimized models for both PCA dataset and non-PCA dataset. Even though results from PCA dataset are acceptable, they consistently lag the performance of the non-PCA dataset. Reason could be the dataset that I was working with

was not ‘fat’ enough, where number of features were not greater than the number of observations. In future exercises, I would still attempt PCA and compare results for PCA and non-PCA datasets.

- In addition to comparing PCA and non-PCA datasets, we could compare the performances for all stages of the feature engineering, including imputations and transformations.

4. Model selection
    - Many models were too expensive to fully optimize
    - Neural networks didn’t perform well with regression – neural networks were slow, produced lackluster results, and were difficult to optimize. They didn’t work well with this regression exercise.
    - It would be interesting to find ways to evaluate models other than duration and metric.

Improvement

I didn’t exhaust all efforts to optimize models. Ideally, I would try all the possible combinations to optimize models. However, optimization efforts turned out to be too computationally expensive, especially for MLP, Keras, Gradient Boosting and XGBoost. For instance, when I tried GridSearchCV with Gradient Boosting, one effort took more than 48 hours on my machine. It was also quite common for my machine to run for more than 10 hours to optimize MLP or Keras algorithms. It might well be possible that these models perform better than Lasso, but I didn’t have the computing power nor the time to find out. Therefore, it would be necessary to leverage AWS to accomplish these model optimization tasks.

I would also be interested in visualizing hyperparameters’ relationship with model performances. Currently, this was accomplished by plotting error terms vs. parameters in K-Folds. If I were to repeat this exercise, I would build a pipeline where I could generalize using visualizations on how parameters affect model performances.

Citation and Sources

Data source and description: https://www.kaggle.com/c/house-prices-advanced-regression-techniques

Comprehensive data exploration with python by Pedro Marcelino: https://www.kaggle.com/pmarcelino/comprehensive-data-exploration-with-python

Stacked Regressions: Top 4% on LeaderBoard: https://www.kaggle.com/serigne/stacked-regressions-top-4-on-leaderboard

Relevant Files and Folders

Code:

- Feature_Engineering.py (output nonPCA and PCA feature engineered datasets)
- Model Optimization GridSearchCV (find optimial hyperparameters for each modeling algos)
- Model Optimization Kfold (give initial idea on how hyperparameters relate with metric for each modeling algos)

Model Performance:

- Model Selection.xlsx (performance summary for the data and modeling algos matrix)

Data Dictionary and Summary:

- Features.xlsx (include feature dictionary and feature metrics)

Data:

- train.csv (from kaggle, used this to feature engineer, develop and verify model performances)
- test.csv (from kaggle, use this to feature engineer and produce output to submit to kaggle)

Visualizations:

- \Feature Distribution (compare distributions before and after transformations)
- \Hyper Parameter Performance (display relationship between model hyperparameters and metrics)
- \Missing Treatment Variable Distribution (compare distribution before and after imputation)